

BÖLÜM 1

ALGORİTMANIN TEMELLERİ

ANAHTAR KAVRAMLAR

Algoritma, Tasarım Tekniği,
Sözde Kod, Akış Şeması,
Zaman Karmaşıklığı, Alan Karmaşıklığı

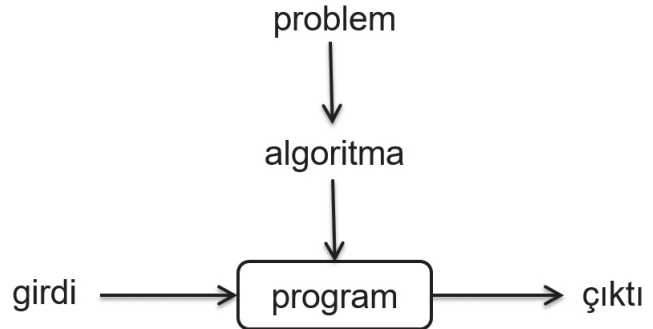
ÖĞRENME HEDEFLERİ

- 1 Algoritma kavramının tanımını bilir
- 2 Algoritmaların sözde kod, akış şeması gibi farklı formlarda nasıl ifade edilebileceğini bilir
- 3 Algoritmanın doğruluğunun ne olduğu ve nasıl gösterilebileceğini bilir
- 4 Algoritmanın performans analizinde kullanılan zaman karmaşıklığı ve alan karmaşıklığı gibi kavramları bilir

GİRİŞ

Bilinen ilk formel örnekleri çok daha eski çağlara dayansa da; kelime olarak ‘algoritma’, 9. yüzyılda yaşamış İranlı matematikçi el-Harizmi’nin isminin Latince karşılığından türetilmiştir. Tanımı üzerine bir mutabakat bulunmasa da; algoritma, belirli bir problemi çözmeye yönelik oluşturulmuş açıkça belirtilmiş adımlar kümesidir denilebilir. Süpermarkette belirli bir ürünü ararken, arkadaşımıza evimizi tarif ederken, ya da internetten aldığımız bir mobilyanın kurulumunu yaparken, aslında farkında olmasak da günlük pratikte sık sık algoritmalara başvurmaktayız.

Yukarıda verilen tanımı bilgisayar bilimlerine uygulayacak olursak; Şekil 1.1’de de gösterildiği gibi algoritma, belirli bir problemin çözümü için tasarlanmış, geçerli veri ya da veri kümesini girdi olarak alan ve sonlu adımda istenen çıktı değerini üreten talimatlar bütünüdür.



Şekil 1.1 Algoritmanın Tanımı (Levitin, 2012)

Bu bölümde ‘algoritma’ kavramının daha iyi anlaşılabilmesi için, algoritmanın temel özellikleri diyebileceğimiz belli bir takım kriterler irdelenecektir.

1. PROBLEMİN ANLAŞILMASI

Verilen bir problem için etkin bir algoritma tasarımında ilk yapılması gereken elbette üzerinde çalışılan problemin iyi bir şekilde anlaşılması olacaktır. Burada problemin anlaşılmasından kasıt, algoritmanın hangi elemanları girdi olarak alacağını, ve neticede hangi sonucu çıktı olarak üreteceğinin açık bir şekilde tespit edilmesidir. Ayrıca; yukarıdaki tanımdan da anlaşılacağı üzere, algoritmanın sadece bazı girdiler için değil, geçerli bütün girdiler için istenen çıktıyı üretmesi gerekmektedir.

Verilen bir tamsayı dizisinin en büyük elemanını bulan bir algoritma tasarladığımızı düşünelim. Tasarlayacağımız algoritma girdi olarak kullanıcıdan belli sayıda tamsayı değeri alacak ve çıktı olarak girilen tamsayıların en büyüğünü üretecektir. Bu problemde olduğu gibi algoritmanın üzerinde çalışacağı girdiyi ve sonuçta üreteceği çıktıyı tespit etmek her problem için bu kadar kolay olmamaktadır. Örneğin verilen iki DNA dizisinin (eşit ya da farklı uzunluklu) benzer

olup olmadıklarını tespit eden bir algoritma tasarladığımızı düşünelim. Tasarladığımız algoritma A, C, G ve T harflerinden oluşan iki kelimeyi girdi olarak alacaktır. Fakat yukarıdaki problem-den farklı olarak bu iki dizinin benzer olup olmadıklarının tespiti için nasıl bir sonuç üreteleceği problem-den doğrudan anlaşılammamaktadır. Algoritma tasarımına başlamadan önce benzerliği tespit etmemize imkan verecek sonuç değerinin belirlenmesi gerekmektedir. Örneğin verilen iki DNA dizisinin en uzun ortak alt dizisinin boyutunun kısa dizinin boyutuna oranı benzerliğin ölçümü için kriter olarak kullanılabilir. Böylece geliştirilen algoritmayı problemin aşağıda verilen örneğine uygularsak; girdide de belirtildiği gibi verilen iki DNA dizisinin en uzun ortak alt dizisi CACT'ın boyutu 4 ve kısa dizinin boyutu 8 olduğundan çıktı değeri 1/2 olacaktır.

girdi : ACCACTGGT, ATCACTTG

çıkıtı : 1/2

2. TASARIM TEKNİĞİNİN BELİRLENMESİ

Verilen bir probleme etkili bir algoritma geliştirmek için yapılması gerekenlerden bir diğeri de algoritma için probleme uygun tasarım tekniğinin belirlenmesi olacaktır. Algoritma tasarım tekniği, bilgisayar bilimlerindeki belirli bir takım problemlere uygulanan ve problem çözümünde izlenmesi gereken adımları belirleyen genel bir yaklaşım şeklinde tanımlanabilir. Bilgisayar bilimlerindeki problemlerin çözümü için geliştirilen algoritmaları, tasarım esnasında izlenen belirli bir takım ortak özellikler ve yaklaşımlar üzerinden tasarım tekniği başlığı altında gruplamak; benzer problemlerin çözümü için daha hızlı ve etkili bir biçimde yeni algoritmalar geliştirebilmemize imkan sağlarken, aynı zamanda algoritmaları üzerinde çalıştığımız problemler yerine algoritma tasarım süreciyle ilintili daha uygun bir kriter üzerinden sınıflandırmamıza da imkan sağlamaktadır. Kitabın ikinci bölümünde kaba kuvvet, azalt & fethet, böl & fethet, ağaçlı yaklaşım gibi bilgisayar bilimlerinde yaygın olarak kullanılan tasarım teknikleri incelenecektir.

Tasarım tekniğine karar verirken, bazı tekniklerin uygulanabilmesi için problemin belirli bir takım şartları sağlaması gerektiği unutulmamalıdır. Örneğin verilen problemin çözümünde 'böl & yönet' tasarım tekniğinin uygulanabilmesi için, problemin alt problemlere (çoğunukla birbirinden bağımsız) bölünebilmesi ve alt problemlerin çözümü ile orjinal problemin çözüm arasında ilişki kurulabilmesi gerekmektedir.



DİKKAT EDELİM

Algoritma gösteriminde hangi yöntem kullanılırsa kullanılsın, algoritmayı oluşturan adımların açık ve sarıh bir biçimde ifade edilmesi gerekmektedir.

3. ALGORİTMANIN NASIL İFADE EDİLECEĞİNİN BELİRLENMESİ

Algoritma tasarlandıktan sonra elbette belirli bir formda ifade edilmesi gerekmektedir. Verilen bir problemin çözümü için geliştirilmiş algoritma konuşma dilinde ifade edilebileceği gibi, algoritma girdilerinin görsel olarak hangi adımlar üzerinden çıktıya dönüştüğünü gösteren akış şeması ya da sözde kod (pseudocode) üzerinden de gösterilebilmektedir. Burada sözde kod konuşma dili ile programlama dilleri arasında köprü vazifesi gören ve algoritmaların ilgili programlama dillerine kolayca aktarılabilmesine olanak sağlayan bir ara formdur. Algoritma gösteriminde hangi yöntem kullanılırsa kullanılsın, algoritmayı oluşturan adımların açık ve sarıh bir biçimde ifade edilmesi gerekmektedir. Bu doğrultuda;

algoritmaları sözde kodla gösterirken, okuyanın işini ve ilgili algoritmanın programlama dillerine aktarılma sürecini kolaylaştırmak amacıyla kodun gerekli yerlerine ‘//’ notasyonunu kullanarak açıklamalar eklemek faydalı olacaktır.

Algoritmaların yukarıda zikredilen formlar üzerinden nasıl ifade edildiğini bir örnek üzerinden anlamaya çalışalım. Şekil 1.2’de yukarıda da bahsi geçen verilen bir tamsayı dizisinin en büyük elemanını bulma problemi için konuşma dili kullanılarak gösterilmiş bir algoritma verilmiştir. Önce algoritmanın girdi ve çıktı değerleri belirtilmiş ve sonrasında alınan girdi değerleri üzerinden algoritmanın nasıl sonuç değerini ürettiğini adımlar şeklinde gösterilmiştir.

Girdi : N elemanlı bir tam sayı dizisi

Çıktı : Dizinin en büyük elemanı

Adım 1: A ve K adıyla iki değişken tanımla ve dizinin ilk elemanını A değişkenine, 2 değerini K değişkenine ata

Adım 2: Dizinin K. elemanı A’dan büyük ise, A’ya dizinin K. elemanını ata ve K’yı bir artır; değilse, K’yı bir artır

Adım 3: K > N ise, algoritmanın çıktısı olarak A değerine dön; değilse, adım 2’ye geri dön

Şekil 1.2 En Büyük Elemanı Bulma Algoritmasının Konuşma Dilinde Gösterimi

Aşağıda algoritmanın 9, 5 ve 12 sayılarından oluşan 3 elemanlı bir tamsayı dizisi üzerinde na-

sıl çalıştığı ve en büyük elemanı nasıl bulduğu adım adım gösterilmiştir.

Adım 1: A = 9, K = 2

Adım 2: Dizinin K. elemanı (yani ikinci) A’dan büyük olmadığından, K = 3

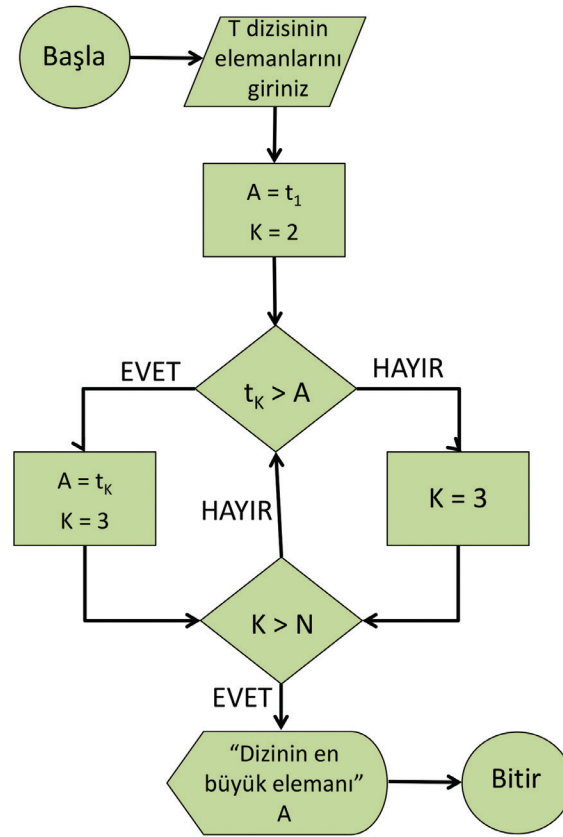
Adım 3: K N’den büyük olmadığından, Adım 2’ye dön

Adım 2: Dizinin K. elemanı (yani üçüncü) A’dan büyük olduğundan, A = 12, K = 4

Adım 3: K N’den büyük olduğundan A’nın değerini yani 12’yi çıktı olarak belirle

Şekil 1.3’de yukarıda konuşma dili kullanılarak gösterilen ve verilen bir tamsayı dizisinin en büyük elemanını bulmaya yarayan algoritmanın akış şeması verilmiştir. Şekilde de görüleceği üzere akış şeması algoritmanın adımlarını belirten kutucuklardan ve adımlar arası geçişleri

gösteren oklardan meydana gelmektedir. Algoritmanın nasıl çalıştığını görsel bir dille anlattığı için diğer alternatifleriyle kıyaslandığında akış şemaları algoritmaların daha anlaşılır bir dille ifadesine imkan sağlamaktadır.



Şekil 1.3 En Büyük Elemanı Bulma Algoritmasının Akış Şeması

Yukarıda da ifade edildiği gibi, sözde kod konuşma dili ile programlama dilleri arasında köprü vazifesi gören ve algoritmaların ilgili programlama dillerine kolayca aktarılabilmesine olanak sağlayan bir ara formdur. Her ne kadar sözde kod programlama dillerindekine benzer ifadeler içerse de, bu ifadelerin bilgisayarlar tarafından anlaşılıp uygulanması mümkün değildir. Dolayısıyla sözde kod ile ifade edilen algoritmaların salt yazıldığı gibi programlama dillerine aktarılması yerine ilgili programlama diline adapte edilmesi gerekmektedir. Sözde kod üzerinden gösterilen algoritmaların programlama dillerine aktarılması diğer alternatiflerine göre daha kolay olduğundan, algoritmaların sözde kod ile ifadesi etkin ve tercih edilen bir yöntemdir. Şekil 1.3’de akış şeması ile gösterilen ve verilen bir tamsayı dizisinin en büyük elemanını bulmaya yarayan algoritmanın sözde kod ile gösterimi Şekil 1.4’de verilmiştir.

```

1  EnBüyük([t1, t2, ..., tN])
2
3  girdi : N elemanlı bir tam sayı dizisi
4  çıktı : Dizinin en büyük elemanı
5
6  // lokal maksimumu tutacak bir A değişkeni tanımlandı
7  // sırasıyla dizinin elemanları tarayabilmek için K değişkeni tanımlandı
8
9  A ← t1
10 K ← 2
11 while K ≤ N
12   // lokal maksimumu tutacak bir A değişkeni tanımlandı
13   if tK > A
14     A ← tK
15     K ← K + 1
16 return A
  
```

Şekil 1.4 En Büyük Elemanı Bulma Algoritmasının Sözde Kodu

Konuşma dilinde olduğu gibi sözde kodla gösterimde de önce algoritmanın girdi ve çıktı bilgisi verilmiştir. Sözde kodla gösterimde farklı notasyonlar kullanılsa da bu kitapta değişkenlere değer atamak için ‘←’ notasyonu kullanılacaktır.

Bunun yanı sıra, okuyanın işini ve ilgili algoritmanın programlama dillerine aktarılma sürecini kolaylaştırmak amacıyla kodun gerekli yerlerine ‘//’ notasyonunu kullanarak açıklamalar eklenmiştir.

Girdi : Başlangıç ve bitiş saatleri verilmiş N tane seminer talebi
Çıktı : Maksimum sayıda seminer içeren bir seminer planı

Adım 1: Başlama saati en erken olan seminer talebini seminer planına ekle

Adım 2: Adım 1’de belirlenen seminer talebini ve bu taleple çakışan seminer taleplerini girdi kümesinden çıkar

Adım 3: Girdi kümesinde seminer talebi kalmadıysa seminer planına dön; değilse, adım 1’e geri dön

Şekil 1.5 Seminer Planlama Algoritması

4. ALGORİTMANIN DOĞRULUĞUNUN İSPATLANMASI

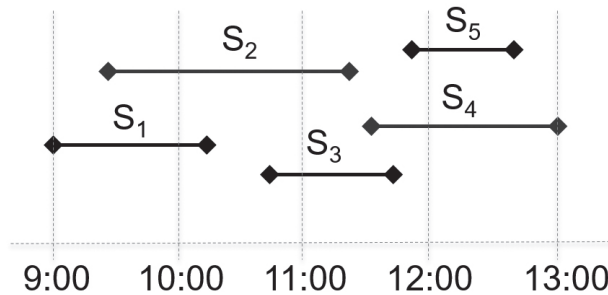
Algoritmalar yukarıda bahsi geçen formlardan biri yoluyla ifade edildikten sonra yapılması gereken algoritmaların doğruluğunun ispatlanması olacaktır. Algoritmaların doğruluğunun ispatıyla neyin kastedildiğini, seminer planı problemi üzerinden anlamaya çalışalım. Seminer planı problemi; tek bir seminer odası için oluşturulmuş başlangıç ve bitiş saatleri belirli bir dizi se-

miner talebi arasından, birbiriyle uyumlu maksimum sayıda semineri içeren bir seminer planı oluşturma şeklinde tanımlanabilir. Burada birbiriyle uyumludan kasıt, oluşturulan planda seminerlerin çakışmamasıdır. Problemi çözümü için Şekil 1.5’de konuşma dili kullanılarak gösterilmiş algoritmayı göz önünde bulunduralım.



DİKKAT EDELİM

Sözde kod ile ifade edilen algoritmaların salt yazıldığı gibi programlama dillerine aktarılması yerine ilgili programlama diline adapte edilmesi gerekmektedir.



Şekil 1.6 Seminer Planı İçin Bir Örneklem

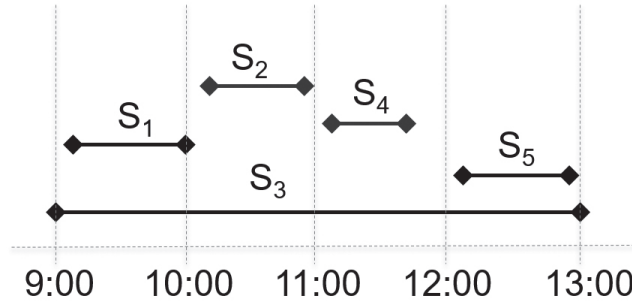
Şekil 1.6'deki örneklem üzerinden yukarıdaki algoritmanın doğruluğunu irdelemeye çalışalım. Örneklemde seminer odası için 9:00 – 13:00 saatleri arasına yerleştirilebilecek 5 talep bulun-

Adım 1: S_1 'i seminer planına ekle
 Adım 2: S_1 'i ve S_1 'le çakışan S_2 'yi girdi kümesinden çıkar
 (girdi kümesinde geriye S_3 , S_4 ve S_5 kaldı)
 Adım 3: Girdi kümesinde seminer talebi kaldığından Adım 1'e dön
 Adım 1: S_3 'ü seminer planına ekle
 Adım 2: S_3 'ü ve S_3 'le çakışan S_4 'ü girdi kümesinden çıkar
 (girdi kümesinde geriye S_5 kaldı)
 Adım 3: Girdi kümesinde seminer talebi kaldığından Adım 1'e dön
 Adım 1: S_5 'i seminer planına ekle
 Adım 2: S_5 'i girdi kümesinden çıkar
 Adım 3: Girdi kümesinde seminer talebi kalmadığından seminer planına dön ($\{S_1, S_3, S_5\}$)

Bununla birlikte, algoritmanın bu örnek üzerinde, ya da bir kaç örneklem üzerinde doğru çalıştığını göstermiş olmak algoritmanın doğruluğunu ispatlamak için yeterli değildir. Örneğin aynı algoritma Şekil 1.7'de gösterilen örneklem üzerinde doğru sonuç üretmeyecektir. Şunu belirtmekte fayda var ki, bu örneklem gibi algoritmanın doğru sonuç üretmediği bir örneklem bulmak algoritmanın yanlışlığını ispatlamak için yeterli olacaktır. Diğer yandan, algoritmanın

maktadır. Aşağıda algoritmanın verilen örneklem üzerinde nasıl çalıştığı ve seminer planını nasıl bulduğu adım adım gösterilmiştir.

aşağıdaki örneklem benzer doğru sonuç üretmediği herhangi bir örneklem bulamıyor olmak da algoritmanın doğruluğunu ispatlamak için yeterli olmayacaktır. Algoritmanın doğruluğunu ispatlamak için yapılması gereken; doğrudan ispat, çelişki yoluyla ispat ya da tümevarım gibi matematiksel ispat yöntemleri kullanarak formel olarak algoritmanın bütün geçerli girdi değerleri için doğru çıktı değerini ürettiğini göstermektir.



Şekil 1.7 Seminer Planı İçin Alternatif Bir Örneklem

5. ALGORİTMANIN ETKİNLİK ANALİZİNİN YAPILMASI

Algoritmalar tasarlandıktan sonra yapılması gereken bir diğer şey de geliştirilen algoritmanın performansının ya da etkinliğinin analizini yapmak olacaktır. Etkinlik analizi algoritmanın pratikte uygulanıp uygulanamayacağına dair fikir verirken, aynı zamanda aynı problem için geliştirilmiş farklı algoritmaları da kıyaslayabilmemize imkan sağlamaktadır. Algoritmalar için iki tür etkinlik analizi göz önünde bulundurulmaktadır:

zaman etkinliğinin analizi (ya da zaman karmaşıklığı) algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğinin belirlenmesi, alan etkinliğinin analizi (ya da alan karmaşıklığı) algoritmanın verilen girdi için istenen sonucu üretirken bellekten ne kadarlık kısma ihtiyaç duyduğu şeklinde tanımlanabilir. Algoritmanın etkinlik analizi Bölüm 5'te detaylıca incelenecektir.

BÖLÜM ÖZETİ

Bu bölümde; belirli bir problemi çözmeye yönelik oluşturulmuş açıkça belirtilmiş adımlar kümesi şeklinde tanımlanabilecek algoritma kavramının tasarım süreci; doğruluk, performans analizi gibi kriterler üzerinden incelenmiştir. Verilen bir problem için etkin bir algoritma tasarımında ilk yapılması gereken elbette üzerinde çalışılan problemin iyi bir şekilde anlaşılması olacaktır. Burada problemin anlaşılmasından kasıt, algoritmanın hangi elemanları girdi olarak alacağını ve neticede hangi sonucu çıktı olarak üreteceğinin açık bir şekilde tespit edilmesidir. Tasarım sürecinde bir sonraki adım; kaba kuvvet, böl & yönet, açgözlü yaklaşım gibi tasarım teknikleri arasından probleme en uygun olanını belirlemek olacaktır. Bunların yanı sıra, elbette geliştirilen algoritmanın konuşma dili, akış şeması ya da sözde kod gibi gösterim yöntemlerinden hangisi kullanılarak ifade edileceğinin belirlenmesi de gerekmektedir. Bölümde algoritmaların bu formlarda nasıl ifade edildikleri örnekler üzerinden anlatılmıştır. Algoritmalar bahsi geçen bu formlardan biri vasıtasıyla ifade edildikten sonra yapılması gereken bir diğer şey, algoritmaların doğruluğunun ispatı olmalıdır. Bölümde algoritmalar için doğruluk kriteri ile neyin kastedildiği örnek üzerinden detaylıca ele alınmıştır. Son olarak algoritma tasarımında atlanmaması gereken bir diğer şey de geliştirilen algoritmanın performansının ya da etkinliğinin analizini yapmak olmalıdır. Etkinlik analizi algoritmanın pratikte uygulanıp uygulanamayacağına dair fikir verirken, aynı zamanda aynı problem için geliştirilmiş farklı algoritmaları da kıyaslayabilmemize imkân sağlamaktadır.

GÖZDEN GEÇİRELİM

1. Algoritma kavramını tanımlayınız.

2. Seminer planı problemi; tek bir seminer odası için oluşturulmuş başlangıç ve bitiş saatleri belirli bir dizi seminer talebi arasından, biririyle uyumlu maksimum sayıda semineri içeren bir seminer planı oluşturma şeklinde tanımlanmaktadır. Buna göre seminer planı probleminin çözümü için geliştirilmiş algoritmanın hangi girdi değerlerini aldığını ve sonuçta hangi çıktı değerini ürettiğini belirleyiniz.

3. Adem N basamaklı bir merdiveni çıkmaya çalışıyor ve her adımda ya tek basamak ya da iki basamak çıkabiliyor. Adem'in merdiveni kaç farklı şekilde çıkabileceğini belirleyen bir algoritmanın hangi girdi değerlerini aldığını ve sonuçta hangi çıktı değerini ürettiğini belirleyiniz.

4. Girilen iki tamsayıdan büyük olanını belirleyen bir algoritma tasarlayınız ve algoritmanın akış şemasını gösteriniz.

5. Verilen bir tamsayı dizisinin elemanlarının toplamını hesaplayan bir algoritma tasarlayınız ve algoritmanın konuşma dilinde gösterimini yazınız.

6. Konuşma dilindeki gösterimi aşağıdaki gibi olan algoritmanın sözde kod ile gösterimini yazınız.

Girdi : N elemanlı bir tamsayı dizisi ve X tamsayısı

Çıktı : Dizi X'i içeriyorsa, dizinin X'e eşit ilk elemanının indeksi; değilse, 0 değeri

Adım 1: K isimli bir değişken tanımla ve 1 değerini K değişkenine ata

Adım 2: Dizinin K. elemanı A'ya eşitse K değerine dön; değilse, K'yı bir artır

Adım 3: $K > N$ ise, algoritmanın çıktısı olarak 0 değerine dön; değilse, adım 2'ye geri dön

7. Sözde kod ile gösterimi aşağıdaki gibi olan algoritmanın konuşma dilindeki gösterimini yazınız.

Büyük($[t_1, t_2, \dots, t_N], X$)

girdi : N elemanlı bir tamsayı dizisi ve X tamsayısı

çıktı : Dizinin X'ten büyük elemanlarının sayısı

// X'ten büyük elemanların sayısını tutacak A değişkeni tanımlandı

// sırasıyla dizinin elemanlarını tarayabilmek için K değişkeni tanımlandı

$A \leftarrow 0$

$K \leftarrow 1$

while $K \leq N$

if $t_K > X$

$A \leftarrow A + 1$

$K \leftarrow K + 1$

return A

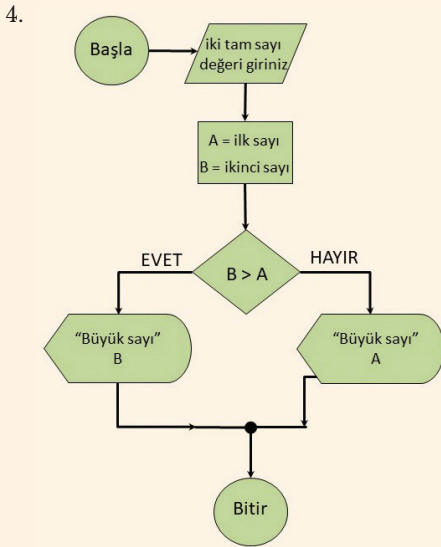
8. Verilen bir tamsayı dizisinin farklı elemanlardan oluşup oluşmadığını belirleyen bir algoritma tasarlayınız ve algoritmanın sözde kod ile gösterimini yazınız.

10. Zaman karmaşıklığı ve alan karmaşıklığı kavramlarını tanımlayınız.

9. Algoritmalar için doğruluk kriterini tanımlayınız.

Yanıt Anahtarı

1. Belirli bir problemi çözmeye yönelik oluşturulmuş açıkça belirtilmiş adımlar kümesidir.
2. Girdi: Başlangıç ve bitiş saatleri verilmiş N tane seminer talebi, Çıktı : Maksimum sayıda seminer içeren bir seminer planı
3. Girdi : Merdivendeki basamak sayısı N, Çıktı : N'nin bir fonksiyonu olarak Adem'in merdivenleri kaç farklı şekilde çıkabileceğinin sayısı



5. Girdi : N elemanlı bir tamsayı dizisi
Çıktı : Dizinin elemanlarının toplamı
Adım 1: A ve K adıyla iki değişken tanımla ve dizinin ilk elemanını A değişkenine, 2 değerini K değişkenine ata
Adım 2: Dizinin K. elemanını A'ya ekle ve K'yı bir artır
Adım 3: $K > N$ ise, algoritmanın çıktısı olarak A değerine dön; değilse, adım 2'ye geri dön
6. Arama($[t_1, t_2, \dots, t_N], X$)
Girdi : N elemanlı bir tamsayı dizisi ve X tamsayısı
Çıktı: Dizi X'i içeriyorsa, dizinin X'e eşit ilk

elemanının indeksi; değilse, 0 değeri
// sırasıyla dizinin elemanları tarayabilmek için K değişkeni tanımlandı
 $K \leftarrow 1$
while $K \leq N$
if $t_K = X$
return K
 $K \leftarrow K + 1$
return 0

7. Girdi: N elemanlı bir tamsayı dizisi ve X tamsayısı
Çıktı: Dizinin X'ten büyük elemanlarının sayısı
Adım 1: A ve K adıyla iki değişken tanımla ve 0 değerini A değişkenine, 1 değerini K değişkenine ata
Adım 2: Dizinin K. elemanı A'dan büyük ise, A'yı 1 artır ve K'yı bir artır; değilse, K'yı bir artır
Adım 3: $K > N$ ise, algoritmanın çıktısı olarak A değerine dön; değilse, adım 2'ye geri dön

8. Ayrık Vadi($[t_1, t_2, \dots, t_N]$)
girdi : N elemanlı bir tamsayı dizisi
çıkı : Dizi farklı elemanlardan oluşuyorsa 'true'; değilse, 'false'
for ($i = 1$ to N)
for ($j = i$ to N)
if $t_i = t_j$
return false
return true

9. Algoritmanın doğruluğu, algoritmanın bütün geçerli girdi değerleri için doğru çıktı değerini üretmesi şeklinde tanımlanmaktadır.

10. Zaman karmaşıklığı algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğinin belirlenmesi, alan karmaşıklığı ise algoritmanın verilen girdi için istenen sonucu üretirken bellekten ne kadarlık kısma ihtiyaç duyduğu şeklinde tanımlanmaktadır.

Kaynakça

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 2

ALGORİTMA TASARIM TEKNİKLERİ

ANAHTAR KAVRAMLAR

Algoritma, Tasarım Tekniği,
Kaba Kuvvet, Azalt & Fethet,
Böl & Fethet, Açgözlü Yaklaşım

ÖĞRENME HEDEFLERİ

1

Tasarım tekniğinin ne olduğunu ve algoritmik problem çözümünde ne ifade ettiğini bilir

2

Kaba kuvvet tasarım tekniğinin nasıl uygulanacağını bilir

3

Azalt & fethet tasarım tekniğinin hangi problem sınıfına nasıl uygulanacağını bilir

4

Böl & fethet tasarım tekniğinin hangi problem sınıfına nasıl uygulanacağını bilir

5

Açgözlü yaklaşım tasarım tekniğinin nasıl uygulanacağını bilir

6

Farklı tasarım tekniklerinden hangisinin verilen probleme daha etkin bir şekilde uygulanabileceğini bilir

GİRİŞ

Algoritma tasarım tekniği, bilgisayar bilimlerindeki belirli bir takım problemlere uygulanan ve problem çözümünde izlenmesi gereken adımları belirleyen genel bir yaklaşım şeklinde tanımlanabilir. Bilgisayar bilimlerindeki problemlerin çözümü için geliştirilen algoritmaları, tasarım esnasında izlenen belirli bir takım ortak özellikler ve yaklaşımlar üzerinden tasarım tekniği başlığı altında gruplamak; benzer problemlerin çözümü için daha hızlı ve etkili bir biçimde yeni algoritmalar geliştirebilmemize imkan sağlarken, aynı zamanda algoritmaları üzerinde çalıştığımız problemler yerine algoritma tasarım süreciyle ilintili daha uygun bir kriter üzerinden sınıflandırmamıza da imkan sağlamaktadır. Bu bölümde kaba kuvvet, azalt & fethet, böl & fethet, ağgözlü yaklaşım gibi bilgisayar bilimlerinde yaygın olarak kullanılan tasarım teknikleri incelenecektir.



1. KABA KUVVET (BRUTE FORCE)

Kaba kuvvet tasarım tekniği kullanılarak geliştirilen algoritmalar (kaba kuvvet algoritmaları) algoritmik problem çözümünde genellikle ilk akla gelen en toy ve en aşık algoritmalardır denilebilir. Bu tasarım tekniğinde problem çözümü için üzerine kafa yorulmuş ve zekice tasarlanmış algoritmalar yerine, çoğunlukla problemin tanımından hareketle geliştirilmiş daha basit ve ilkel algoritmalar tercih edildiği için, kaba kuvvet algoritmalarının performansı diğer tasarım tekniklerine nazaran daha kötüdür. Bu nedenle kaba kuvvet etkili bir tasarım tekniği olarak değerlendirilmemektedir. Kaba kuvvet tasarım tekniği ile algoritmaların nasıl geliştirildiği örnekler üzerinden anlatılacaktır.

Verilen n tane tamsayıyı küçükten büyüğe sıralama problemini ele alalım. Bu problem için

akla ilk gelen basit ve aşık çözüm dizinin küçük elemanlarını bulup dizinin ön sıralarına taşımak olacaktır. Kabaca, önce dizinin en küçük elemanı bulunup dizinin ilk sırasına, sonrasında dizinin en küçük ikinci elemanı bulunup dizinin ikinci sırasına yerleştirilecektir. Bu şekilde dizinin 3., 4., ..., n . elemanları bulunup sırasıyla dizinin 3., 4., ..., n . sırasına yerleştirilecek ve nihayetinde küçükten büyüğe sıralı bir dizi elde edilecektir. Burada bahsi geçen algoritma sıralama problemi için geliştirilmiş kaba kuvvet algoritmalarından biri olup literatürde seçmeli sıralama (selection sort) ismiyle yer almaktadır. Seçmeli sıralama algoritmasının konuşma dilindeki gösterimi Şekil 2.1'de verilmiştir.

Girdi : N tane sıralanabilir elemandan oluşan bir dizi
Çıktı : Dizinin elemanlarının küçükten büyüğe sıralanışı

Adım 1: K isimli bir değişken tanımla ve 1 değerini K değişkenine ata

Adım 2: Dizinin en küçük K . elemanını bul ve dizinin K . sırasına yerleştir, ve K 'yı bir artır

Adım 3: Eğer $K > N$ ise, diziyi dön; değilse, adım 2'ye geri dön

Şekil 2.1 Seçmeli Sıralama Algoritmasının Konuşma Dilinde Gösterimi

Aşağıda bir kaba kuvvet sıralama algoritması olan seçmeli sıralama algoritmasının verilen bir örneklem üzerinde nasıl çalıştığı ve sıralı diziyi

nasıl bulduğu adım adım gösterilmiştir. Burada algoritma girdi olarak [7, 12, 5] dizisini almıştır.

Adım 1: $K = 1$

Adım 2: Dizinin en küçük elemanı 5'i finalde çıktı olarak dönülecek dizinin ilk sırasına yerleştir ve $K = 2$

Adım 3: $K > N$ olmadığından Adım 2'ye dön

Adım 2: Dizinin en küçük ikinci elemanı 7'yi finalde çıktı olarak dönülecek dizinin ikinci sırasına yerleştir ve $K = 3$

Adım 3: $K > N$ olmadığından Adım 2'ye dön

Adım 2: Dizinin en küçük üçüncü elemanı 12'yi finalde çıktı olarak dönülecek dizinin üçüncü sırasına yerleştir ve $K = 4$

Adım 3: $K > N$ olduğundan yeni sıralamaya dön ([5, 7, 12])

Şimdi de dizgi eşleştirme (string matching) problemini ele alalım. Dizgi eşleştirme problemi, verilen bir metnin verilen bir dizgiyi içerip içermediğini belirleme problemi olarak tanımlan-

maktadır. Bu problem için ilk akla gelen çözüm, metindeki dizgi ile aynı boyuttaki bütün kelimeleri belirlemek ve sonrasında dizginin bunlardan herhangi biri ile eşleşip eşleşmediğine bakmak

olacaktır. Tasarlanan bu kaba kuvvet algoritması; eğer dizgi kelimelerden biri ile eşleşiyorsa ilgili kelimenin ilk sembolünün metindeki indeksine dönecek, içermiyorsa sıfıra dönecektir. Kimi zaman, ana problemin bir parçası olan daha küçük problemler için algoritmalar geliştirip bunları alt

yordam olarak kullanmak algoritma tasarımında esneklik sağlayacaktır. Bu problemde de önce verilen eşit uzunluklu iki dizginin eşleşip eşleşmediğini belirleyen bir algoritma tasarlanacak ve yukarıda açıklanan kaba kuvvet algoritmasında alt yordam olarak kullanılacaktır.

Girdi : N tane sembolden oluşan iki dizgi
Çıktı : Dizgiler aynıysa 1, değilse 0

Adım 1: K isimli bir değişken tanımla ve 1 değerini K değişkenine ata

Adım 2: İki dizginin K. elemanı aynıysa, K'yı bir artır; değilse, 0'a dön

Adım 3: Eğer $K > N$ ise, 1'e dön; değilse, adım 2'ye geri dön

Şekil 2.2 Eşleştirme Algoritmasının Konuşma Dilinde Gösterimi

Şekil 2.2'de verilen eşit uzunluklu iki dizginin eşleşip eşleşmediğini (aynı olup olmadığını) belirleyen algoritmanın konuşma dilindeki gösterimi, Şekil 2.3'de ise bu algoritmayı alt yordam

olarak kullanan kaba kuvvet dizgi eşleştirme algoritmasının konuşma dilindeki gösterimi verilmiştir.

Girdi : N tane sembolden oluşan bir metin ve M tane sembolden oluşan bir dizgi
Çıktı : Metin dizgiyi içeriyorsa, dizgiyle eşleşen metindeki ilk kelimenin ilk harfinin metindeki indeksi; değilse, 0

Adım 1: Metindeki dizgi ile aynı boyutlu bütün kelimeleri belirle (toplamda $N - M + 1$ tane)

Adım 2: K isimli bir değişken tanımla ve 1 değerini K değişkenine ata

Adım 3: Adım 1'de belirlenen K. kelime ile dizgi üzerinde, Şekil 1.8'de verilen eşleştirme algoritmasını çalıştır; eşleştirme algoritmasının çıktısı 1 ise K değerine dön, değilse K'yı artır

Adım 4: Eğer $K > N - M + 1$ ise, 0'a dön; değilse, adım 3'e geri dön

Şekil 2.3 Dizgi Eşleştirme Algoritmasının Konuşma Dilinde Gösterimi

Aşağıda dizgi eşleştirme problemi için geliştirilmiş Şekil 2.3'deki kaba kuvvet algoritmasının verilen bir örneklem üzerinde nasıl çalıştığı adım

Adım 1: ATA, TA_, A_B, _BA, BAK

Adım 2: $K = 1$

Adım 3: ATA ile BAK kelimesi için eşleştirme algoritması 0 değerini döneceğinden $K = 2$

Adım 4: $K > N - M + 1$ olmadığından Adım 3'e dön (burada $N - M + 1 = 7 - 3 + 1 = 5$)

Adım 3: TA_ ile BAK kelimeleri için eşleştirme algoritması 0 değerini döneceğinden $K = 3$

Adım 4: $K > N - M + 1$ olmadığından Adım 3'e dön

Adım 3: A_B ile BAK kelimeleri için eşleştirme algoritması 0 değerini döneceğinden $K = 4$

adım gösterilmiştir. Burada algoritma girdi olarak 'ATA_BAK' metni ile 'BAK' dizgisini almıştır.

Adım 4: $K > N - M + 1$ olmadığından Adım 3'e dön

Adım 3: _BA ile BAK kelimeleri için eşleştirme algoritması 0 değerini döndüğünden $K = 5$

Adım 4: $K > N - M + 1$ olmadığından Adım 3'e dön

Adım 3: BAK ile BAK kelimeleri için eşleştirme algoritması 1 değerini döndüğünden algoritma $K = 5$ değerine döner.

Son olarak da sırt çantası (knapsack) problemi ele alalım. Sırt çantası problemi; M kapasitesine sahip bir çanta ve ağırlıkları ve değerleri belli çantaya koyulabilecek N eşya verildiğinde, çantayı ağırlıkları toplamı kapasiteyi aşmayacak şekilde değerleri toplamı maksimum olan eşyalarla doldurma problemi olarak tanımlanmaktadır. Bu gibi problemler için kaba kuvvet algoritmalarında genellikle yapılan bütün olası durumları belirlemek ve bu olası durumları tek tek inceleyerek hangisinin verilen kısıtı sağlayan

optimum çözüm olduğunu bulmak olmaktadır. Bu problem için de geliştireceğimiz kaba kuvvet algoritması önce verilen eşyalar üzerinden olası tüm eşya kümelerini belirleyecek ve sonrasında sırasıyla bu kümeleri inceleyerek hangi kümenin ağırlık kısıtını sağlayan değerleri toplamı en yüksek eşya kümesi olduğunu bulacaktır. Sırt çantası problemi için tasarladığımız kaba kuvvet algoritmasının konuşma dilindeki gösterimi Şekil 2.4'de verilmiştir.

Girdi : M kapasiteli bir sırt çantası ve sırt çantasına konulabilecek ağırlığı ve değeri verilmiş N tane eşya
Çıktı : Değerleri toplamı maksimum olan ağırlıkları toplamı M 'yi aşmayan eşya kümesi

Adım 1: Olası bütün eşya kümelerini belirle (toplamda 2^N tane)

Adım 2: K , A ve B isimli üç değişken tanımla ve 1 değerini K değişkenine 0 değerini A ve B değişkenlerine ata

Adım 3: Adım 1'de belirlenen K . eşya kümesi için; kümedeki eşyaların ağırlıkları toplamı M 'yi aşmıyorsa ve kümedeki eşyaların değerleri toplamı A 'dan büyükse, eşyaların değerleri toplamını A 'ya ve K 'yi B 'ye ata. K 'yi 1 artır

Adım 4: Eğer $K > 2^N - 1$ ise, adım 1'de belirlenen B . eşya kümesine ve A değerine dön; değilse, adım 3'e geri dön

Şekil 2.4 Sırt Çantası Algoritmasının Konuşma Dilinde Gösterimi

Aşağıda sırt çantası problemi için geliştirilmiş Şekil 2.4'deki kaba kuvvet algoritmasının verilen bir örneklem üzerinde nasıl çalıştığı adım adım gösterilmiştir. Burada algoritma girdi olarak 50

ağırlık kapasiteli bir sırt çantası ve ağırlıkları sırasıyla (10, 20, 30) ve değerleri sırasıyla (60, 100, 120) olan 3 eşya almıştır.

Adım 1: {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}

Adım 2: $K = 1$, $A = 0$, $B = 0$

Adım 3: Birinci kümedeki eşyaların ağırlıkları toplamı 10, 50'yi aşmadığından ve eşyaların değerleri toplamı 60, 0'dan büyük olduğundan $A = 60$, $B = 1$. $K = 2$

Adım 4: $K > 2^N - 1$ olmadığından Adım 3'e dön (burada $2^N - 1 = 7$)

Adım 3: İkinci kümedeki eşyaların ağırlıkları toplamı 20, 50'yi aşmadığından ve eşyaların değerleri toplamı 100, 60'dan büyük olduğundan $A = 100$, $B = 2$. $K = 3$

Adım 4: $K > 2^N - 1$ olmadığından Adım 3'e dön

Adım 3: Üçüncü kümedeki eşyaların ağırlıkları toplamı 30, 50'yi aşmadığından ve eşyaların değerleri toplamı 120, 100'den büyük olduğundan $A = 120$, $B = 3$. $K = 4$

Adım 4: $K > 2^N - 1$ olmadığından Adım 3'e dön

Adım 3: Dördüncü kümedeki eşyaların ağırlıkları toplamı 30, 50'yi aşmadığından ve eşyaların değerleri toplamı 160, 120'den büyük olduğundan $A = 160$, $B = 4$. $K = 5$

Adım 4: $K > 2^N - 1$ olmadığından Adım 3'e dön

Adım 3: Beşinci kümedeki eşyaların ağırlıkları toplamı 40, 50'yi aşmadığından ve eşyaların değerleri toplamı 180, 160'dan büyük olduğundan $A = 180$, $B = 5$. $K = 6$

Adım 4: $K > 2^N - 1$ olmadığından Adım 3'e dön

Adım 3: Altıncı kümedeki eşyaların ağırlıkları toplamı 50, 50'yi aşmadığından ve eşyaların değerleri toplamı 220, 180'den büyük olduğundan $A = 220$, $B = 6$. $K = 7$

Adım 4: $K > 2^N - 1$ olmadığından Adım 3'e dön

Adım 3: Yedinci kümedeki eşyaların ağırlıkları toplamı 60, 50'yi aştığından $K = 8$

Adım 4: $K > 2^N - 1$ olduğundan algoritma $\{2, 3\}$ eşya kümesine ve $A = 220$ değerine dönecektir

Yukarıda da belirttiğimiz gibi performansının diğer alternatiflerine nazaran çok daha kötü olması sebebiyle kaba kuvvet etkili bir tasarım tekniği olarak değerlendirilmemektedir. Bununla birlikte, diğer alternatiflerinin aksine kaba kuvvet çok daha geniş yelpazedeki problemlere

uygulanabilmektedir. Ayrıca; performansın çok da önemli olmadığı algoritmanın yalnızca küçük boyutlu ya da sınırlı sayıdaki girdilere uygulandığı durumlarda, kaba kuvvet tasarım tekniğini kullanmak tasarımcıyı daha etkili algoritmalar geliştirme zahmetinden kurtaracaktır.

2. AZALT & FETHET (DECREASE & CONQUER)

Eğer verilen bir problemin çözümü ile problemin daha küçük boyutlu alt probleminin çözümü arasında bir ilişki varsa, ve alt problemin çözümü bu ilişki üzerinden genişletilerek ana problemin çözümü elde edilebiliyorsa; akıllıca olan doğrudan ana problemin çözümü için uğraşmaktansa, görece daha kolay olan alt problemi çözmek ve sonrasında bu çözümü kullanarak ana problemin çözümünü elde etmek olacaktır. Azalt & fethet tasarım tekniğinde yapılan; verilen problem eğer yukarıda ifade edilen özelliği sağlıyorsa, ana problemi çözmek yerine girdi boyutunu küçülterek bir alt problem elde etmek ve görece daha kolay bu alt problem için çözüm üretmektir. Sonrasında elde edilen çözüm gözlemlenen ilişki üzerinden ana problemin çözümüne genişletilecektir.

Verilen iki sayının en büyük ortak bölenini bulma problemini ele alalım. Verilen iki M, N sa-

yıları için; büyük sayının N olduğu kabul edilirse, $OBE(N, M) = OBE(M, N \bmod M)$. Bu eşitlik bize ana problemin çözümü (N ve M sayılarının en büyük ortak böleni) ile daha küçük boyutlu alt problemin çözümünün (M ve $N \bmod M$ sayılarının en büyük ortak böleni) aynı olduğunu, yani ana problemin çözümü ile daha küçük boyutlu alt problemin çözümü arasında doğrudan ilişki olduğunu söylemektedir. Dolayısıyla bu ilişkiden istifade ederek verilen probleme azalt & fethet tasarım yöntemini uygularsak Şekil 2.5'de sözde kod ile gösterimi verilen algoritmayı elde ederiz. Dikkat edilirse bu özyinemeli algoritma ile her adımda girdi boyutu belli oranda azaltılarak algoritmanın daha hızlı ve etkili bir biçimde çıktı üretmesi sağlanmıştır. Bu algoritma literatürde Öklid algoritması olarak yer almaktadır.

```

1  Öklid(N, M)
2
3  girdi :  $N$  ve  $M$  sayıları
4  çıktı :  $EBOB(N, M)$ 
5
6  // sayılardan  $N$ 'nin büyük olduğu varsayılmaktadır
7  if  $M = 0$ 
8      return  $N$ 
9  else
10     return  $\text{Öklid}(M, N \bmod M)$ 

```

Şekil 2.5 Öklid Algoritmasının Sözde Kodu

Öklid algoritmasının verilen girdi üzerinde nasıl çalıştığını bir örneklem üzerinden anlamaya çalışalım. Örneklemde kullanılacak sayılar 24 ve 18 olsun. İkinci sayı 0'dan farklı olduğundan algoritma EBOB(24, 18)'i bulmak yerine Öklid (18, 6) kodu ile daha küçük boyutlu alt problem olan EBOB(18, 6)'yı hesap edecektir. Sonraki adımda

yine ikinci sayı 0'dan farklı olduğundan algoritma EBOB(18, 6)'yı bulmak yerine Öklid (6, 0) kodu ile daha küçük boyutlu alt problemin çözümünü bulmaya çalışacaktır. Öklid(6, 0) için, ikinci sayı sıfır olduğundan algoritma ilk sayıya dönecektir. Yani algoritma EBOB(24, 18)'i 6 olarak bulacaktır.

Girdi : Bir birine tıpatıp benzeyen ve bir tanesi hafif olmak üzere geri kalanları aynı ağırlıkta olan N top ve eşit kollu bir terazi
Çıktı : Hafif top

Adım 1: N tekse, bir topu kenara ayırıp kalan topları iki eşit paya ayırıp bu payları terazinin kefelerine yerleştir; değilse topları iki eşit paya ayırıp bu payları terazinin kefelerine yerleştir

Adım 2: Eğer paylar eşit ağırlıktaysa, kenarda bırakılan topu çıktı olarak belirle

Adım 3: Paylar eşit ağırlıkta değilse ve hafif olan kefedeki tek bir top varsa, bu topu çıktı olarak belirle

Adım 4: Paylar eşit ağırlıkta değilse ve hafif olan kefedeki birden fazla top varsa, bu topları girdi kabul et ve adım 1'e dön

Şekil 2.6 Hafif Top Algoritmasının Konuşma Dilinde Gösterimi

Şimdi de hafif topu bulma problemini ele alalım. Hafif topu bulma problemi, birbirine tıpatıp benzeyen ve bir tanesi hafif olmak üzere geri kalanları aynı ağırlıkta olan N top ve eşit kollu bir terazi verildiğinde, hafif topu minimum tartma işlemi ile bulma problemi olarak tanımlanmaktadır. N sayısının çift olduğunu varsayalım. Topların yarısını terazinin sağ kefesine, diğer yarısını da sol kefesine koyarsak, ve sol kefedeki toplar daha hafif gelirse, aradığımız hafif top sol kefedekilerden biridir diyebiliriz. Burada dikkat ederseniz, başlangıçta verilmiş olan N tane topun hafif olanıyla, sol kefedeki N/2 tane topun hafif

olanı aynı olacaktır. Diğer bir ifadeyle, N boyutlu ana problemin çözümü ile sol kefedeki N/2 boyutlu alt problemin çözümü arasında doğrudan bir ilişki olacaktır. Dolayısıyla bu ilişkiden istifade ederek verilen probleme azalt & fethet tasarım yöntemini uygularsak Şekil 2.6'da konuşma dili ile gösterimi verilen algoritmayı elde ederiz. Dikkat edilirse burada da yukarıdaki algoritmaya benzer biçimde özyinemeli olarak her adımda girdi boyutu belli oranda azaltılıp algoritmanın daha hızlı ve etkili bir biçimde çıktı üretmesi sağlanmıştır.

3. BÖL & FETHET (DIVIDE & CONQUER)

Tasarım teknikleri içerisinde en bilineni ve yaygın olarak kullanılanı böl & fethet tasarım tekniğidir. Azalt & fethet tekniğinde olduğu gibi, eğer verilen bir problemin çözümü ile problemin daha küçük boyutlu alt problemlerinin çözümleri arasında bir ilişki varsa, ve alt problemlerin çözümü birleştirilerek ana problemin çözümü elde

edilebiliyorsa; böl & fethet tasarım tekniğinde de yapılan ana problemi doğrudan çözmek yerine, problemi alt problemlere ayırıp özyinelemeli bir şekilde alt problemleri çözmektir. Sonrasında bu çözümler belirlenen ilişki üzerinden birleştirip ana problemin çözümü elde edilmektedir.

```

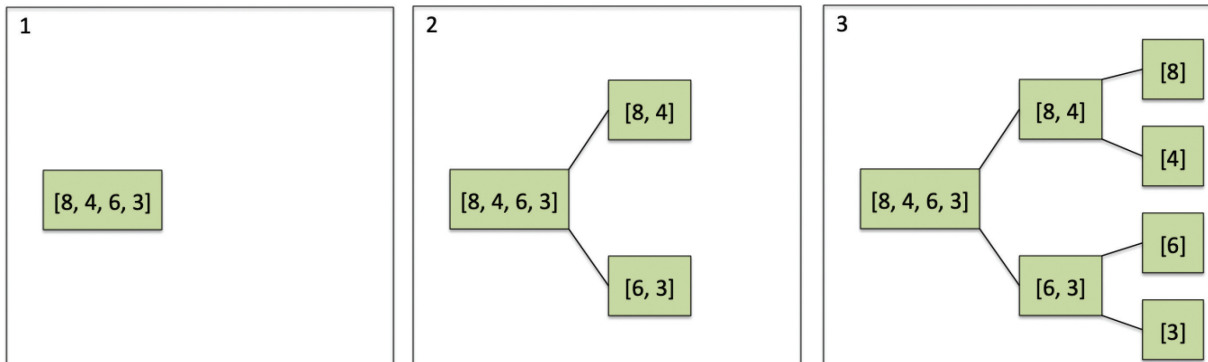
1  EnKucuk([ $t_i, t_{i+1}, \dots, t_j$ ])
2
3  girdi : N elemanlı bir tam sayı dizisi
4  çıkıtı : Dizinin en küçük elemanı
5
6  // dizide tek bir eleman kaldıysa algoritma o elemana döner
7  if  $i = j$ 
8      return  $t_i$ 
9  else
10      $mid \leftarrow (i + j)/2$ 
11     // dizinin ilk yarısının en küçük elemanı
12      $kucuk1 \leftarrow \text{EnKucuk}([t_i, \dots, t_{mid}])$ 
13     // dizinin ikinci yarısının en küçük elemanı
14      $kucuk2 \leftarrow \text{EnKucuk}([t_{mid+1}, \dots, t_j])$ 
15     if  $kucuk1 < kucuk2$ 
16         return  $kucuk1$ 
17     else
18         return  $kucuk2$ 

```

Şekil 2.7 En Küçük Elemanı Bulma Algoritmasının Sözde Kodu

Böl & fethet nasıl uygulandığını, verilen N elemanlı bir sayı dizisinin en küçük elemanını bulma problemi üzerinden anlamaya çalışalım. Burada problemi doğrudan çözmek yerine; problemi, sırasıyla dizinin ilk N/2 elemanının en küçük elemanını bulma ve son N/2 elemanının en küçük elemanını bulma problemleri şeklinde iki alt probleme bölelim ve bu alt problemlerin çözümlerini bulalım. Dikkat edilirse bu alt problemlerin çözümlerinden küçük olanı,

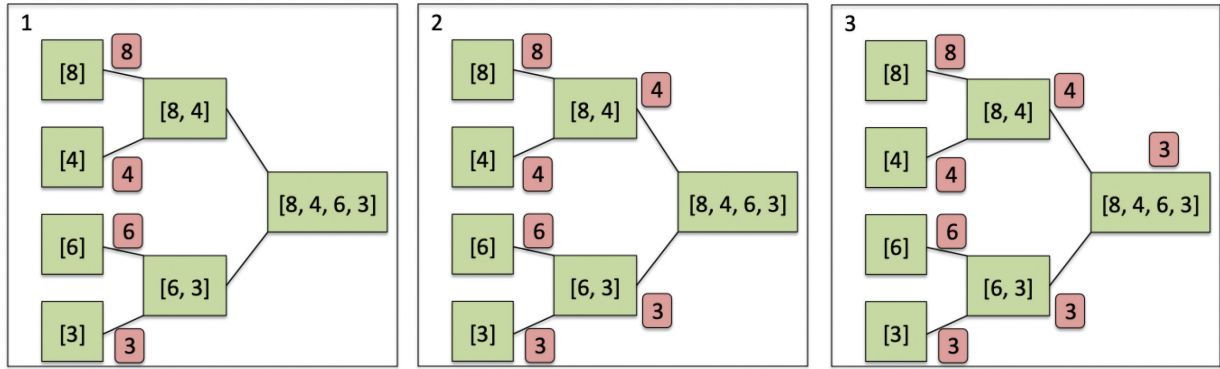
yani dizinin sırasıyla ilk yarısının en küçük elemanı ile ikinci yarısının en küçük elemanından küçük olanı ana problemin çözümü olacaktır. Burada geliştireceğimiz algoritma alt problemleri özyinelemeli bir şekilde çözecektir. Yani, ana problemde uyguladığımız böl ve fethet stratejisi alt problemlerin çözümünde de kullanılacaktır. Problemin çözümü için tasarladığımız böl & fethet algoritmasının sözde kod gösterimi Şekil 2.7'de verilmiştir.



Şekil 2.8 Problemin Alt Problemlere Ayrılması

Böl & fethet tasarım tekniğini daha iyi anlayabilmek adına, Şekil 2.7'de verilen algoritmanın [8, 4, 6, 3] örneklemini üzerinde adım adım nasıl çalıştığını inceleyelim. Algoritma Şekil 2.8'de gösterildiği gibi problemi alt problemlere, verilen alt problemleri de daha küçük boyutlu alt problemlere ayıracaktır. Bu süreç özyinelemeli

olarak alt problemler tek bir eleman içerene kadar devam edecektir. Sonrasında ise, Şekil 2.9'da gösterildiği gibi alt problemlerin çözümleri birleştirilerek daha büyük alt problemlerin çözümleri elde edilecektir. Yine benzer şekilde bu süreç de özyinelemeli olarak ana problemin çözümü elde edilinceye kadar devam edecektir.



Şekil 2.9 Alt Problemlerin Çözümlerinin Birleştirilmesi

Şimdi de büyük sayıların çarpımı problemini ele alalım. Günümüzde bazı uygulamalar, özellikle bir takım modern kriptografik teknikler 300 basamaklı gibi çok büyük sayıların çarpımını gerektirmektedir. Bu boyuttaki sayıların pratik bir şekilde çarpılabilmesi modern kriptografideki süreçleri hızlandıracaktır. Böl & fethet tasarım tekniğinin bu probleme nasıl uygulanabileceğini, 6 basamaklı 123456 ve 654321 sayılarının çarpımı gibi basit bir örnek üzerinden anlamaya çalışalım. Burada doğrudan çarpım yerine, daha etkili bir yöntem elde etmek adına sayıları $123456 = 123000 + 456$ ve $654321 = 654000 + 321$ şeklinde iki parçaya ayrılmış olarak ele alalım. Verilen sayıların çarpımını bu formlar üzerinden yeniden oluşturursak: $123456 \times 654321 = 123 \times 654 \times 10^6$

+ $(123 \times 321 + 456 \times 654) \times 10^3 + 456 \times 321$ denklemini elde etmiş oluruz. Görüleceği üzere, burada 6 basamaklı iki sayının çarpımı problemi 4 adet 3 basamaklı iki sayının çarpımı alt problemine indirgenmiştir. Genelleyecek olursak, N basamaklı A ve B sayıları verildiğinde, sayılar önce $A = A_1 \times 10^{N/2} + A_2$ ve $B = B_1 \times 10^{N/2} + B_2$ şeklinde N/2 basamaktan oluşan iki parçaya ayrılmış olarak ele alınacak ve bu parçaların çarpımları $A_1 \times B_1$, $A_1 \times B_2$, $A_2 \times B_1$ ve $A_2 \times B_2$ hesaplanacaktır. Sonra bu çözümler $A \times B = A_1 \times B_1 \times 10^N + (A_1 \times B_2 + A_2 \times B_1) \times 10^{N/2} + A_2 \times B_2$ denklemi üzerinden birleştirilecek ve ana problemin çözümü bulunmuş olacaktır. Şekil 2.10'da büyük sayıların çarpımı problemi için geliştirilmiş böl & fethet algoritmasının sözde kod ile gösterimi verilmiştir.



DİKKAT EDELİM

Azalt & fethet ve böl & fethet tasarım tekniklerinin probleme uygulanabilmesi için problemin alt probleme indirgenebilmesi veya alt problemlere ayrılabilir olması ve alt problem veya problemlerin çözümüyle ana problemin çözümü arasında ilişki kurulabilmesi gerekmektedir.


```

1  Carpim(A, B, N)
2
3  girdi : N basamaklı iki sayı
4  çıkıtı : Sayıların çarpımı
5
6  If n = 1
7      return AxB
8  // sayılar iki parçaya ayrılmış olarak ele alınır
9   $A \leftarrow A_1 \times 10^{N/2} + A_2$ 
10  $B \leftarrow B_1 \times 10^{N/2} + B_2$ 
11 // verilen parçalar üzerinden oluşturulan alt problemler
12 // hesaplanır
13  $X \leftarrow \text{Carpim}(A_1, B_1, N/2)$ 
14  $Y \leftarrow \text{Carpim}(A_1, B_2, N/2)$ 
15  $Z \leftarrow \text{Carpim}(A_2, B_1, N/2)$ 
16  $T \leftarrow \text{Carpim}(A_2, B_2, N/2)$ 
17 // alt problemlerin çözümleri birleştirilir
18 return  $X \times 10^N + (Y + Z) \times 10^{N/2} + T$ 

```

Şekil 2.10 Böl & Fethet Çarpım Algoritmasının Sözde Kodu

4. AÇGÖZLÜ YAKLAŞIM (GREEDY APPROACH)

Verilen bir problemi açgözlü yaklaşımla çözerken; çözümün her adımında, o andaki mevcut ve problemin istediği şartları sağlayan lokal çözümlerden en uygun olanı tercih edilip ana problemin çözümüne eklenmektedir. Böylelikle adım adım lokal çözümler üzerinden ana problemin çözümü oluşturulmaktadır. Bu tasarım tekniğinde dikkat edilmesi gereken nokta, çözümün her

adımında tasarım tekniğinin adından da anlaşılacağı üzere aç gözlü bir şekilde hareket edildiği ve lokaldeki çözümün bizi nihayetinde ana problemin istediği optimum çözüme ulaştırıp ulaştırmayacağını bilinmediği gerçeğidir. Bu tasarım tekniği genellikle optimizasyon problemleri için tercih edilmektedir.

```

1  Kasiyer([t1, t2, ..., tN], M)
2
3  girdi : M kuruş para üstü ve bozuk para çeşitleri [t1, t2, ..., tN]
4  çıkıtı : Minimum sayıda bozuk para
5
6  // Bozuk para çeşitleri t1 > t2 > ... > tN şeklinde sıralıdır
7  // A para üstü K ise toplam bozuk para miktarı için
8  // oluşturulmuş değişkenlerdir
9   $A \leftarrow M$ 
10  $j \leftarrow 1$ 
11  $K \leftarrow 0$ 
12 while j ≤ N
13     if tj ≤ A
14          $A \leftarrow A - t_j$ 
15          $K \leftarrow K + 1$ 
16     else
17          $j \leftarrow j + 1$ 
18 return K

```

Şekil 2.11 Kasiyer Algoritmasının Sözde Kodu

Bu tasarım tekniğinde nasıl algoritma geliştirildiğini kasiyer problemi ile anlamaya çalışalım. Kasiyer problemi, para üstü için gerekli olan

minimum sayıda bozuk parayı bulma problemi olarak tanımlanmaktadır. Burada kullanılacak bozuk para çeşitleri 25, 10, 5 ve 1 kuruş

olarak belirlenmiş olsun. Örneğin 62 kuruş para üstü için, bunu 6 tane 10 kuruş ve 2 tane 1 kuruş kullanarak verebileceğimiz gibi, 1 tane 25 kuruş, 3 tane 10 kuruş ve 7 tane 1 kuruş kullanarak da verebiliriz. İlk çözümde toplamda 8 tane bozuk para kullanılırken, ikinci çözümde 11 tane bozuk para kullanılmıştır. Peki optimum çözüm hangi bozuk paralardan oluşuyor olacaktır? Burada açgözlü yaklaşım tekniğini takip ettiğimizde yapacağımız ilk şey mümkün olan en büyük bozuk para ile çözümü oluşturmaya başlamak olacaktır. Dolayısıyla 62 kuruş para üstü için ilk kullanacağımız bir adet 25 kuruş olacaktır. Kalan 37 kuruş

için aynı strateji izlersek yine bir 25 kuruş daha kullanmamız gerekecek. Kalan 12 kuruş için, kullanılacak en yüksek bozuk para çeşidi 10 kuruş olduğundan çözüme bir 10 kuruş eklememiz gerekecek. Son olarak kalan 2 kuruş için de 2 adet 1 kuruş kullanmamız gerekecek. Dolayısıyla finalde elde edeceğimiz optimum çözüm 2 adet 25 kuruş, 1 adet 10 kuruş ve 2 adet 1 kuruş , yani toplamda 5 adet bozuk para içerecektir. Burada anlattığımız açgözlü yaklaşım tekniği ile geliştirilmiş kasiyer algoritmasının sözde kod ile gösterimi Şekil 2.11'de verilmiştir.

BÖLÜM ÖZETİ

Bu bölümde kaba kuvvet, azalt&fethet, böl&fethet, açgözlü yaklaşım gibi bilgisayar bilimlerinde yaygın olarak kullanılan tasarım teknikleri anlatılmıştır. Algoritma tasarım tekniği, bilgisayar bilimlerindeki belirli birtakım problemlere uygulanan ve problem çözümünde izlenmesi gereken adımları belirleyen genel bir yaklaşım şeklinde tanımlanabilir. Kaba kuvvet tasarım tekniğinde problem çözümü için üzerine kafa yorulmuş ve zekice tasarlanmış algoritmalar yerine, çoğunlukla problemin tanımından hareketle geliştirilmiş daha basit ve ilkel algoritmalar tercih edilmektedir. Bu yüzden kaba kuvvet algoritmalarının performansı diğer tasarım tekniklerine nazaran daha kötüdür. Bölüm içerisinde bu tasarım tekniğinde nasıl algoritma geliştirildiği farklı problem sınıflarından seçilmiş örneklerle detaylıca incelenmiştir. Azalt & fethet ve böl & fethet tasarım tekniklerinin probleme uygulanabilmesi için problemin alt probleme indirgenebilmesi veya alt problemlere ayrılabilir olması ve alt problem veya problemlerin çözümüyle ana problemin çözümü arasında ilişki kurulabilmesi gerekmektedir. Azalt & fethet tasarım tekniğinde ana problem için çözüm bulmak yerine problemin alt problemi için çözüm geliştirilmekte ve sonrasında bu çözüm ana problemin çözümüne genişletilmektedir. Benzer şekilde böl & fethet tasarım tekniğinde de ana problem alt problemlere ayrılmakta ve özyinelemeli bir biçimde alt problemlerin çözümleri birleştirilerek ana problemin çözümü elde edilmektedir. Kaba kuvvet tekniğinde olduğu gibi bu tasarım teknikleri de örnekler üzerinden detaylı bir şekilde ele alınmıştır. Bölümde son olarak açgözlü yaklaşım tekniğinde algoritma geliştirme sürecinin nasıl yürütüldüğü bir örnek üzerinden açıklanmıştır.

GÖZDEN GEÇİRELİM

1. N elemanlı bir tamsayı dizisi ve bir X sayısı verildiğinde, kaba kuvvet tasarım tekniğini kullanarak dizinin toplamları X'e eşit olan iki sayı içerip içermediğini bulan bir algoritma geliştiriniz ve algoritmanın konuşma dilindeki gösterimini yazınız.

2. Verilen bir X tamsayısı ve N pozitif tamsayısı için, kaba kuvvet tasarım tekniğini kullanarak "X'in N. kuvvetini hesaplayan bir algoritma geliştiriniz ve algoritmanın sözde kod ile gösterimini yazınız.

3. Verilen N tane harften oluşan bir kelime için, kaba kuvvet tasarım tekniğini kullanarak bu kelimenin A harfi ile başlayan anlamlı ya da anlamsız kaç tane alt kelime içerdiğini bulan bir algoritma geliştiriniz ve algoritmanın konuşma dilindeki gösterimini yazınız.

4. Aşağıda sözde kod ile gösterimi verilen ve verilen bir sıralı tamsayı dizisinin ve verilen bir X sayısını içerip içermediğini belirleyen Arama algoritması hangi tasarım tekniği ile geliştirilmiştir belirleyiniz?

Arama (T, i, j, X)

Girdi : i başlangıç indeksi ve j bitiş indeksi olmak üzere $j - i + 1$ elemandan oluşan T tamsayı dizisi ve X tamsayısı

Çıktı : Dizi X'i içeriyorsa, dizinin X'e eşit ilk elemanının indeksi; değilse, 0

//başlangıç indeksi bitiş indeksinden büyükse, dizi X'i içermiyordur

if $i > j$

return 0

//dizinin ortası belirlenir

mid = $(i + j) / 2$

if $X = t_{mid}$

return mid

//dizi sıralı olduğu için, X dizinin ortasındaki elemandan küçükse, X dizinin sol yarısında aranır

elseif $X < t_{mid}$

Arama(T, i, mid-1, X)

//X dizinin ortasındaki elemandan büyükse, X dizinin sağ yarısında aranır

else

Arama(T, mid+1, j, X)

5. Verilen bir problemin çözümü için algoritma geliştirilirken böl & fethet tasarım tekniğinin nasıl uygulandığını belirtiniz.

6. Böl & fethet tasarım tekniğini kullanarak verilen N elemanlı bir tamsayı dizisinin en büyük elemanının bulan bir algoritma geliştiriniz ve algoritmanın sözde kod ile gösterimini yazınız.

7. Aşağıda sözde kod ile gösterimi verilen, para üstü ve kullanılabilecek bozuk para çeşitleri verildiğinde, para üstü için gerekli olan minimum sayıda bozuk parayı bulan Kasiyer algoritmasının doğruluğunu ([10, 6, 1], 24) örneklemini üzerinden irdeleyiniz.

Kasiyer($[t_1, t_2, \dots, t_N], M$)

Girdi : M kuruş para üstü ve bozuk para çeşitleri $[t_1, t_2, \dots, t_N]$

Çıktı : Minimum sayıda bozuk para

// Bozuk para çeşitleri $t_1 > t_2 > \dots > t_N$ şeklinde sıralıdır

// A para üstü K ise toplam bozuk para miktarı için oluşturulmuş değişkenlerdir

A $\leftarrow$ M

j $\leftarrow$ 1

K $\leftarrow$ 0

while j $\leq$ N

if $t_j \leq A$

A $\leftarrow$ A - t_j

K $\leftarrow$ K + 1

else

j $\leftarrow$ j + 1

return K

8 – Verilen bir X tamsayısı ve N pozitif tamsayısı için, azalt & fethet tasarım tekniğini kullanarak X'in N. kuvvetini hesaplayan bir algoritma geliştiriniz ve algoritmanın sözde kod ile gösterimini yazınız.

9. Konuşma dilindeki gösterimi aşağıdaki şekilde olan ve verilen bir T tamsayı dizisinin verilen bir X elemanını içerip içermediğini belirleyen kaba kuvvet tasarım tekniği ile geliştirilmiş algoritmanın (T = [6, 7, 8, 9], X = 9) örneklemini üzerinde nasıl sonuç ürettiğini adım adım gösteriniz.

Girdi : N elemanlı bir T tamsayı dizisi ve X tamsayısı

Çıktı : Dizi X'i içeriyorsa, dizinin X'e eşit ilk elemanının indeksi; değilse, 0 değeri

Adım 1: K isimli bir değişken tanımla ve 1 değerini K değişkenine ata

Adım 2: Dizinin K. elemanı X'e eşitse K değerine dön; değilse, K'yı bir artır

Adım 3: K > N ise, algoritmanın çıktısı olarak 0 değerine dön; değilse, adım 2'ye geri dön

10. Aşağıda sözde kod ile gösterimi verilen ve verilen iki büyük sayının çarpımını bulan böl & yönet tasarım tekniğinde geliştirilmiş Carpim algoritmasındaki alt problemlerin sayısı görüleceği üzere 4'tür. Algoritmayı daha az alt problem içerecek şekilde geliştirerek daha iyi performanslı bir algoritma elde ediniz.

Carpim(A, B, N)

Girdi : N basamaklı iki sayı

Çıktı : Sayıların çarpımı

If n = 1

return Ax B

// sayılar iki parçaya ayrılmış olarak ele alınır

A $\leftarrow$ $A_1 \times 10^{N/2} + A_2$

B $\leftarrow$ $B_1 \times 10^{N/2} + B_2$

// verilen parçalar üzerinden oluşturulan alt problemler hesaplanır

X $\leftarrow$ Carpim($A_1, B_1, N/2$)

Y $\leftarrow$ Carpim($A_1, B_2, N/2$)

Z $\leftarrow$ Carpim($A_2, B_1, N/2$)

T $\leftarrow$ Carpim($A_2, B_2, N/2$)

// alt problemlerin çözümleri birleştirilir

return $X \times 10^N + (Y + Z) \times 10^{N/2} + T$

Yanıt Anahtarı

1. Girdi : N elemanlı bir tamsayı dizisi ve bir X tamsayısı
Çıktı : Dizi toplamı X'e eşit iki eleman içeriyorsa, bu sayılar; değilse, 0
Adım 1: Dizinin olası bütün iki elemanlı alt kümelerini belirle
(toplamda $C(N,2) - N$ 'nin ikili kombinasyonlarının sayısı - tane)
Adım 2: K isimli bir değişken tanımla ve 1 değerini K değişkenine ata
Adım 3: Adım 1'de belirlenen K. alt küme için; kümedeki elemanların toplamı X'e eşitse K. alt kümenin elemanlarına dön; değilse, K'yı 1 artır
Adım 4: Eğer $K > C(N,2)$ ise, 0 değerine dön; değilse, adım 3'e geri dön
2. **Kuvvet(X, N)**
girdi : X tamsayısı ve N pozitif tamsayısı
çıktı : X^N
 $A \leftarrow 1$
 $K \leftarrow 1$
while $K \leq N$
 $A \leftarrow A \cdot X$
 $K \leftarrow K + 1$
return A
3. Girdi : N tane harften oluşan bir kelime
Çıktı : Kelimenin içerdiği A ile başlayan alt kelimelerin sayısı
«Adım 1: Kelimenin bütün alt kelimelerini belirle.»
Adım 2: K ve A isimli iki değişken tanımla, 1 değerini K değişkenine ve 0 değerini A değişkenine ata
Adım 3: Adım 1'de belirlenen K. alt kelime için; alt kelime A ile başlıyorsa A'yı 1 artır ve K'yı 1 artır, değilse K'yı 1 artır
Adım 4: Eğer $K > 2^N$ ise, A değerine dön; değilse, adım 3'e geri dön
4. Arama algoritmasının geliştirilmesinde azalt & fethet tasarım tekniği kullanılmıştır. Dikkat edilirse algoritmanın her adımında girdi boyuta yarıya düşürülerek algoritmanın daha etkin bir şekilde sonuç üretmesi sağlanmaktadır.
5. Önce problemin alt problemleri belirlenir ve ana problemler belirlenen alt problemlere bölünür
- Ana problemi çözmek yerine görece daha kolay alt problemler özyinelemeli bir biçimde çözülür
- Alt problemlerin çözümleri birleştirilerek ana problemin çözümü elde edilir
6. **EnBuyuk([t₁, t₂, ..., t_j])**
girdi : N elemanlı bir tamsayı dizisi
çıktı : Dizinin en büyük elemanı
// dizide tek bir eleman kaldıysa algoritma o elemana döner
if $i = j$
 return t_i
else
 $mid \leftarrow (i + j)/2$
 // dizinin ilk yarısının en büyük elemanı
 $buyuk1 \leftarrow \text{EnBuyuk}([t_1, \dots, t_{mid}])$
 // dizinin ikinci yarısının en büyük elemanı
 $buyuk2 \leftarrow \text{EnBuyuk}([t_{mid+1}, \dots, t_j])$
if $buyuk2 < buyuk1$
 return $buyuk1$
else
 return $buyuk2$
7. Örneklem üzerinde Kasiyer algoritması çalıştırıldığında, algoritma ilk olarak mümkün olan en yüksek bozuk para çeşidi 10 kuruşla çözüme başlayacak ve finaldeki çıktı kümesine iki 10 kuruş ekleyecektir. Sonrasında kalan 4 kuruş için bozuk para çeşitlerinden 10 kuruş ve 6 kuruş kullanılamayacağından, algoritma bozuk para çeşitlerinden 1 kuruşla yoluna

devam edecek ve çıktı kümesine 4 tane 1 kuruş ekleyecektir. Yani algoritma sonunda elde edilen çözüm iki adet 10 kuruştan ve 4 adet 1 kuruştan, yani toplamda 6 adet bozuk paradan oluşacaktır. Halbuki, 24 kuruş para üstü 4 adet 6 kuruş kullanılarak da verilebilirdi. Dolayısıyla Kasiyer algoritması bu örneklem için optimum çözüm üretmemiştir. Her ne kadar bu algoritma bazı bozuk para çeşitleri kümesi için doğru sonuç üretse de, bütün bozuk para çeşitleri düşünüldüğünde doğruluk kriterini sağlamamaktadır.

8. Kuvvet2(X, N)

girdi : X tamsayısı ve N pozitif tamsayısı
 çıktı : X^N
 if $N = 0$
 return 1
 else
 return X.Kuvvet2(X, N-1)

9. Adım 1: $K = 1$

Adım 2: Dizinin ilk elemanı 6 9'a eşit olmadığından $K = 2$
 Adım 3: $K > N$ olmadığından Adım 2'ye dön
 Adım 2: Dizinin ikinci elemanı 7 9'a eşit olmadığından $K = 3$
 Adım 3: $K > N$ olmadığından Adım 2'ye dön
 Adım 2: Dizinin üçüncü elemanı 8 9'a eşit olmadığından $K = 4$

Adım 3: $K > N$ olmadığından Adım 2'ye dön

Adım 2: Dizinin dördüncü elemanı 9 9'a eşit olduğundan $K = 4$ değerine dön

10. Çarpım algoritmasında alt problemlerin çözümünü birleştirirken kullanılan $X \times 10^N + (Y + Z) \times 10^{N/2} + T$ denklemi; $X = A_1 \times B_1$, $Y = A_2 \times B_2$ ve $Z = (A_1 + A_2) \times (B_1 + B_2)$ olmak üzere, $X \times 10^N + (Z - X - Y) \times 10^{N/2} + Y$ şeklinde düzenlenebilir. Böylelikle aynı problem yalnızca 3 alt problem üzerinden çözülebilecektir. Yeni elde edilen algoritmanın sözde kodu aşağıdaki gibi olacaktır.

YeniCarpim(A, B, N)

girdi : N basamaklı iki sayı

çıktı : Sayıların çarpımı

If $n = 1$

 return $A \times B$

// sayılar iki parçaya ayrılmış olarak ele alınır

$A \leftarrow A_1 \times 10^{N/2} + A_2$

$B \leftarrow B_1 \times 10^{N/2} + B_2$

// verilen parçalar üzerinden oluşturulan alt problemler hesaplanır

$X \leftarrow \text{YeniCarpim}(A_1, B_1, N/2)$

$Y \leftarrow \text{YeniCarpim}(A_2, B_2, N/2)$

$Z \leftarrow \text{YeniCarpim}(A_1 + A_2, B_1 + B_2, N/2)$

// alt problemlerin çözümleri birleştirilir

return $X \times 10^N + (Z - X - Y) \times 10^{N/2} + Y$

Kaynakça

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 3

DİZİLER VE BAĞLI LİSTELER

ANAHTAR KAVRAMLAR

Dizi, İşaretçi, Bağlı Liste,
Dairesel Bağlı Liste, Çift Yönlü Bağlı Liste

ÖĞRENME HEDEFLERİ

1

Tek boyutlu ve iki boyutlu dizileri tanımlayabilir ve dizi elemanlarına değer atayabilir

2

Diziler üzerinde gezinebilir ve dizi elemanları üzerinde işlem yapabilir

3

Bağlı listeleri tanımlayabilir ve kullanabilir

4

Bağlı listelerde kullanılan düğümler için bellekte dinamik olarak yer ayrılabilir

5

Bağlı listeler üzerinde gezinebilir, listeye ekleme yapabilir ve listeden eleman çıkarabilir

6

Farklı bağlı liste tiplerini bilir ve onlar üzerinde işlem yapabilir

GİRİŞ

Bir problemi çözmek için kullanılan algoritmaların verimli bir şekilde çalışabilmesi gerekir. Programda kullanılan verilere en hızlı şekilde ulaşmak, veriler içerisinde arama yapmak ve veriler üzerinde gerçekleşen işlemleri başarılı bir şekilde tamamlamak istenen bir durumdur. Bu durumu sağlayabilmek ve verileri belirli kurallar çerçevesinde organize etmek için veri yapıları kullanılır. Veri yapıları aynı zamanda veri üzerinde gerçekleşen temel işlemleri tanımlar. Farklı veri yapıları farklı işlemler ve problemler için uygun olabilir. Bu bölümde diziler ve bağlı listeler anlatılacaktır. Diziler ve bağlı listeler temel veri yapılarıdır ve aynı zamanda ileriki bölümlerde yer alan farklı veri yapılarının tanımlanması ve uygulanmasında kullanılacaklardır.



1. VERİ YAPILARI

Bu bölümde öncelikli olarak veri yapıları kavramı açıklanacak, veri yapıları üzerinde gerçekleştirilen temel işlemler ifade edilecektir. Veri yapısı, bir programda kullanılan verilerin belirli mantıksal kurallar çerçevesinde belirli temel işlemleri uygulayabilmek amacıyla organize edilmiş durumu olarak tanımlanabilir. Veri yapıları doğrusal ve doğrusal olmayan olmak üzere iki grupta değerlendirilebilir. Bir veri yapısında elemanlar ardışık bir düzende yer alıyorsa bu yapılar doğrusal aksi takdirde doğrusal olmayan olarak kabul edilirler. Diziler, bağlı listeler, kuyruklar ve yığınlar doğrusal, ağaçlar ve grafikler doğrusal olmayan veri yapılarına örnek olarak verilebilirler. Veri yapılarındaki temel işlemler aşağıdaki gibi listelenebilir:

Gezinme (Traversal): Bir veri yapısındaki tüm elemanlara belirli bir sıra ve düzen içerisinde erişme işlemidir. Bu gezinme kapsamında ilgili yapının elemanları üzerinde belirli işlemler uygulanabilir. Örneğin veri yapısının elemanları ekrana yazdırılabilir.

Ekleme (Insertion): Bir veri yapısına yeni bir eleman yerleştirme işlemidir. Veri yapısının özelliğine göre ekleme işlemi farklılık gösterebilir. Burada temel amaç ekleme işleminden önce geçerli kurallara sahip veri yapısının ekleme işleminden sonra yeni bir elemanla geçerli kuralları sağlamaya devam etmesidir. Her veri yapısının elemanları arasında tanımlanmış kendine has kuralları mevcuttur. Bu kuralların ekleme ve silme benzeri işlemlerde bozulmaması gerekir.

Silme (Deletion): Bir veri yapısından bir elemanı çıkartma işlemidir. Burada silme işlemi veri yapısının özelliğine göre iki farklı şekilde olabilir. Birincisi silinen elemanın kullanılmak amacıyla veri yapısından uzaklaştırılmasıdır. İkincisi ise artık ihtiyaç duyulmayan bir elemanın veri yapısından çıkartılması işlemidir. Silme işleminin amacı ne olursa olsun bu işlem sonrasında

veri yapısı geçerli kuralları sağlayan durumunu bir eksik elemanla sağlamaya devam etmelidir. Bazı durumlarda ekleme ve silme işlemlerinden sonra veri yapısının karakterine göre belirli düzeltme operasyonları gerçekleştirilebilir.

Arama (Search): Herhangi bir değerin bir veri yapısı içerisinde yer alıp almadığının araştırılması işlemidir. Aranılan değer yapı içerisinde bulunduğunda, değerin yapı içerisindeki yeri (bellek adresi veya indisi) çevrilir. Ancak arama işlemi sonucunda ilgili değer bulunamazsa arama işlemi başarısız olarak kabul edilir ve bu durumu ifade etmek için NULL veya -1 benzeri bir değer çevrilir.

Sıralama (Sorting): Bir veri yapısını oluşturan elemanların artan veya azalan düzende sıralanması işlemidir.

Birleştirme (Merge): Aynı türden iki veri yapısının birleştirilme işlemidir. Bu işlem sonucunda tek bir veri yapısı tüm elemanlara sahip olur. Bu veri yapısının geçerli tüm kuralları sağlaması beklenir.

Yukarıda ifade edilen işlemler farklı veri yapıları için farklı şekilde gerçekleştirilirler. Veri yapısının karakterine göre bazı işlemlerden sonra düzeltme işlemleri gerekebilir. Bazı veri yapılarını oluşturmak için kendileri de birer veri yapısı olarak kabul edilen diziler veya bağlı listeler kullanılır. Bazı veri yapıları hem dizi hem de bağlı listeler ile beraber gerçekleştirilebilir. Her iki yöntemin de avantajları ve dezavantajları vardır.

Her bir veri yapısı üzerinde yapılan işlemler çalışma süresi ve gerektirdiği bellek alanı açısından karşılaştırılabilir.

Öncelikli olarak bu bölümde, diziler ve bağlı listeler yapıları detaylı olarak yer alacaktır. Sonraki bölümlerde kuyruklar, yığınlar, ağaçlar ve hash tabloları anlatılacaktır.

2. DİZİLER

Bir programda, bellekte ayrılan bir alan, değişken olarak kullanılabilir. Programdaki amaçlarına bağlı olarak değişkenler, farklı veri tiplerinden

tanımlanabilirler. Her veri tipinin bellekte kapladığı alan (kullandığı bayt sayısı) farklıdır. Örneğin *char* veri tipi 1 bayt kullanırken *double* veri

tipi 8 baytlık bir alana sahiptir. Kullanılan programlama diline ve programın çalıştırıldığı makineye bağlı olarak veri tipleri için ayrılan alanlar farklılık gösterebilirler. Burada ifade edilen veri tipleri kullanılarak tanımlanan değişkenler sadece bir değer tutmak için oluşturulurlar. Ancak bazı durumlarda sadece bir değişken programdaki ihtiyacı karşılamayabilir. Örneğin, bir derste sınıftaki tüm öğrencilerin aldığı final notlarına programımızda erişmek için aynı veri tipini öğrenci sayısı kadar tekrar edecek şekilde kullanabilmemiz gerekir. Diziler bu tür problemlere çözüm getirmek için tanımlanmış veri yapılarıdır. Diziler de farklı veri tipleri kullanılarak tanımlanabilirler. Bir dizi değişkeni tanımlamak için de-

a[0]	a[1]	a[2]	a[3]	a[4]
13	22	37	-4	5

Şekil 3.1. Tek boyutlu 5 elemanlı bir tamsayı dizisi

Programın çalışması süresince [] işleci dizinin ismi ile beraber kullanılarak dizinin elemanlarına ulaşılabilir. Bir dizideki indisler 0 ile başlar. Dolayısı ile dizinin n. elemanına erişmek için (n-1) indisi kullanılır. **a[2] = 10**; ifadesi ile dizinin 3. elemanına 10 değeri atanmıştır. Aynı şekilde aşağıdaki ifade ile kullanıcıdan, dizinin 2. elemanına değer okunmaktadır.

```
cin>>a[1];
```

Dizinin ismi sabit bir işaretçi olarak kabul edilir ve program içerisinde gösterdiği adres hiçbir şekilde değiştirilemez. Bu değer aynı zamanda dizinin ilk elemanının adresidir. Bir dizinin herhangi bir elemanına erişmek için dizi alt simgeleme işleci kullanılabilir. Aynı zamanda dizi ismi ile beraber sapma (offset) gösterimi ile de ilgili dizi elemanına ulaşılabilir.

Aşağıdaki kod parçası ile dizinin 3. elemanı iki farklı şekilde yazdırılmaktadır:

```
cout<<a[2]<<endl;
cout<<*(a+2)<<endl;
```

Bir dizinin herhangi bir elemanına indis kullanılarak doğrudan ulaşmak mümkündür. Bu dizilerin en önemli avantajlarından birisidir. Diziler

değişkenin veri tipini, ismini ve dizinin boyutunu tanımlama esnasında belirlememiz gerekir.

```
int a[5];
```

ifadesi 'a' isminde 5 elemana sahip bir tamsayı dizisi oluşturmak için kullanılır. Bir dizinin elemanları bellekte ardışık olarak yer alır. Bellekte dizinin elemanları arasında herhangi bir boşluk yer almaz.

Dizinin elemanlarına dizi tanımlanırken veya sonrasında değer atanabilir.

```
int a[5]={13,22,37,-4,5};
```

ifadesi ile 'a' dizisi tanımlanırken, dizinin elemanları atanmıştır (Şekil 3.1).

üzerinde gezinme işlemi bir *for* veya *while* döngüsü yardımı ile basit bir şekilde gerçekleştirilebilir. Döngü değişkeni dizi indisi olarak kullanılıp ilgili dizi elemanına erişim sağlanıp istenilen işlem gerçekleştirilir.

Aşağıdaki kod parçası ile 5 elemanlı bir dizinin tüm elemanlarına 10 değeri eklenmektedir:

```
for(int i=0; i<=4; i++)
    a[i]=a[i]+10;
```

Bir diziye yeni bir eleman eklemek için dizinin boyutunun yeterli olması gerekmektedir. Eğer dizinin tüm elemanlarının değeri var ise yeni bir eleman ekleme işlemi mümkün olmayacaktır. Ekleme işlemi dizinin ilk boş elemanına gerçekleştirilir. Dolayısıyla bir değişken ile programda dizinin ilk boş elemanının yeri tutulabilir. Bu değişken 'top' olarak isimlendirilmiş olsun. Ekleme işlemi aşağıdaki ifade ile gerçekleştirilir:

```
a[top]=newValue;
top++;
```

Değer atama işleminden sonra 'top' değişkeninin değeri bir artırılarak boş alanı göstermeye devam etmesi sağlanır.

Bazı durumlarda dizinin belirli bir indisine ekleme yapmak gerekebilir. Mevcut elemanları kaybetmemek için tüm dizi elemanları eğer boş alan var ise sağa kaydırılır. İlgili indis yeni bir eleman eklemek için hazırlanır.

Bir diziden bir eleman silmek için 'top' değişkeninin değerinin bir azaltılması yeterlidir.

top--;

Ancak silinecek elemanı dizinin son elemanı değil ise silinen elemandan başlamak üzere tüm elemanlar sağdan sola bir eleman kaydırılır. 'top' değişkeninin değeri bir azaltılır.

2.1. İki Boyutlu Diziler

Bazı problemlerde, programda kullanacağımız verinin tablo olarak ifade edilmesi daha uygun olur. Örneğin bir derste belirli sayıda öğrencinin, belirli sayıda sınavdan aldığı notlar bir tablo

ile gösterilebilir (Şekil 3.2). Bu tür durumlarda iki boyutlu diziler kullanılır. İki boyutlu dizilerde, dizi değişkenimiz satır ve sütunlara sahiptir. Her iki boyut da 0 indisi ile başlar.

	S1	S2	S3
Ö1	60	70	30
Ö2	70	50	20
Ö3	45	80	25
Ö4	50	35	40

Şekil 3.2. 4 öğrencinin 3 farklı sınavdan aldığı notlar

Şekil 3.2 de yer alan tabloyu göstermek için iki boyutlu bir dizi tanımlayabiliriz.

int d[4][3];

Tanımlanan bu dizinin elemanları, tek boyutlu dizilerde olduğu gibi tanımlama sırasında veya daha sonra kullanıcıdan okunarak atanabilir. Programın çalışma sürecinde bireysel olarak da dizinin elemanlarına değer verilebilir.

Tanımlama sırasında, aşağıdaki şekilde ilgili tablonun değerleri iki boyutlu diziye atanabilir.

int d[4][3]={60, 70, 30}, {70, 50, 20}, {45, 80, 25}, {50, 35, 40};

İki boyutlu bir dizinin elemanlarına satır ve sütun indisleri kullanarak erişilir. Örneğin yukarıda tanımlanan 'd' dizisinin elemanlarını ekrana yazdırmak için aşağıdaki kod parçası kullanılabilir.

```
for(int i=0; i<=4; i++)
{
    for(int j=0; j<=4; j++)
        cout<<d[i][j]<<' ';
    cout<<endl;
}
```



ARAŞTIRILIM

İki boyutlu bir dizi değişkenini bir fonksiyona gönderme işlemini araştırılım. Dizinin tüm elemanlarını veya tek bir elemanını gönderme seçeneklerini değerlendirelim.

2.2. Bağlı Listeler

Bir diziyi tanımlamak için bellekte ardışık olarak yeterli alana sahip olmak gerekir. Dizinin elemanları, elemanlar arası boşluk olmadan sıralı olarak belleğe yerleştirilirler. Ayrıca bir dizinin tanımlandığı zaman belirlenen boyutu programın çalışması sırasında değiştirilemez. Bazen programın başlangıcında değişkenler belirlenirken gerekli boyut konusunda yeterli bilgiye sahip olunamaz. Bu dezavantajları ortadan kaldırmak amacıyla bağlı listeler kullanılır. Bağlı listeler

oluşturulduğu zaman önceden boyut bilgisine sahip olunması gerekmez. Ayrıca bağlı listeyi oluşturan elemanlar belleğin farklı alanlarında yer alabilirler. Birbirini takip eden bellek alanları bağlı listelerde kullanılmak zorunda değildir.

Bağlı listelerde listenin her bir elemanını göstermek için *struct* yapısı kullanılır. *struct* ile oluşturulan bu elemanlara düğüm ismi verilir. Şekil 3.3'teki *struct* tanımlaması bir bağlı listede kullanılacak düğümler için oluşturulmuştur.

```
1 struct node
2 {
3     int value;
4     struct node *ptr;
5 }
```

Şekil 3.3. Bağlı liste düğümü için *struct* tanımı

Burada tanımlanan *struct* ile oluşturulan düğümde değer olarak sadece bir tamsayı tutulmaktadır. Ancak farklı bağlı listelerde burada kullanılan tamsayı yerine farklı nesneler kullanılabilir. Buradaki tanımlamada yer alan *struct node* işaretçisi bağlı listedeki her bir düğüm ta-

rafından, listede yer alan bir sonraki düğümün adresini göstermek amacıyla kullanılır. Listedeki son düğümün 'ptr' işaretçisinin değeri bir sonraki düğüm söz konusu olmadığı için NULL olarak tanımlanır.



Şekil 3.4. Tek yönlü bağlı liste

Yukarıda, tanımı verilen *struct* yapısı ile oluşturulan örnek bir tek yönlü bağlı liste verilmiştir (Şekil 3.4). Bu liste dört düğümden oluşmakta en son düğümün 'ptr' işaretçisi NULL değerini göstermektedir. Bu bağlı listeyi kullanmak için sadece ilk düğümün adresini tutmak yeterli olur. Bu değer genellikle 'head' veya 'start' işaretçisi olarak isimlendirilir. Bağlı listelerde istenilen değere ulaşmak için tüm listedeki diğer elemanların üzerinde gezinmek gerekir. Dolayısıyla dizilerde olduğu gibi doğrudan istenilen düğüme ulaşmak söz konusu değildir. Ancak bağlı listelerin bellek kullanımı daha verimlidir. Bağlı

listelerde, listedeki eleman sayısının önceden belirlenmesi zorunlu değildir. Bağlı listeye yeni bir düğüm ekleneceği zaman, bellekten düğüm için yer ayrılır. Herhangi bir düğüm bağlı listeden silindiğinde ayrılan alan belleğe geri verilir.

struct node *head=new struct node;

ifadesi ile bellekten yer ayrılır ve *struct node* cinsinden tanımlanan 'head' işaretçisine ilgili alanın adresi atanır.

Şekil 3.5'teki *display* fonksiyonu bir bağlı listedeki tüm değerleri sırasıyla ekrana yazdırır.

```
1 void display (struct node *head)
2 {
3     struct node *temp;
4     temp=head;
5     while (temp!=NULL)
6     {
7         cout << temp->value << ' ';
8         temp=temp->ptr;
9     }
10    cout << endl;
11 }
```

Şekil 3.5. Bağlı listenin elemanlarını yazdırma fonksiyonu

Bu fonksiyon, bağlı listede yer alan ilk düğümü gösteren adresi parametre olarak alır. Fonksiyon içinde geçici bir düğüm işaretçisine (*temp*) bu adres değeri atanır. Bir *while* döngüsü yardımı ile tüm bağlı listenin elemanları ekrana yazdırılır. Bu döngünün içinde, *temp=temp->ptr*; ifadesi

si ile işaretçinin bir sonraki düğümü göstermesi sağlanır. 'temp' işaretçisi en son düğümü gösterdiği zaman bu düğümün değeri ekrana yazdırılır ve *temp=temp->ptr*; ifadesi ile 'temp' değişkeni NULL değerini alır. Bu şekilde *while* döngüsü sonlanır.



DİKKAT EDELİM

Programın çalışması sırasında, bir düğümü gösteren, NULL değerine sahip bir işaretçinin elemanlarına erişmeye çalışmak, hata üretir.

Şekil 3.6'da verilen *insert* fonksiyonu bağlı listenin başına yeni bir düğüm yerleştirir.

```
1 struct node *insert (struct node *head)
2 {
3     struct node *newNode;
4     int newValue;
5     newNode = new struct node;
6     cin >> newValue;
7     newNode->value = newValue;
8     newNode->ptr = head;
9     head = newNode;
10    return head;
11 }
```

Şekil 3.6. Bağlı listeye yeni bir düğüm ekleme fonksiyonu



DİKKAT EDELİM

Bağlı listeye yeni bir düğüm eklemek için bellekte yer ayırmak gerekir.

Fonksiyon öncelikle 'newNode' isminde yeni bir düğüm oluşturur. Bu yeni düğüm için gerekli alan *new* fonksiyonu ile ayrılır. Kullanıcıdan 'newValue' isminde yeni bir tamsayı okunur ve yeni düğümün 'value' değişkenine atanır. Yeni düğümün işaretçisi de önceki 'head' düğümünü gösterir. Yeni eklenen düğüm bağlı listenin yeni

'head' düğümü olur ve fonksiyon bu değeri çevirir. Fonksiyonun dönüş tipi bir *struct node* işaretçisidir.

Şekil 3.7'de verilen fonksiyon bir bağlı listenin sonuna yeni bir düğüm eklemektedir.

```

1 void InsertEnd (struct node *head)
2 {
3     struct node *newNode;
4     struct node *temp
5     int newValue;
6
7     temp=head
8     newNode=new struct node;
9     cin>>newValue;
10    newNode->value=newValue;
11    newNode->ptr=NULL;
12
13    while(temp->ptr!=NULL)
14        temp=temp->ptr;
15
16    temp->ptr=newNode;
17 }

```

Şekil 3.7. Bağlı listenin sonuna yeni bir düğüm ekleme fonksiyonu

Yukarıdaki fonksiyonda yine yeni bir düğüm oluşturulur. Bu düğümü bağlı listenin sonuna eklemek için listenin sonuna ulaşmak gerekir. Bu da 'temp' isminde geçici bir *struct node* işaretçisi ile sağlanır. Listenin ilk düğümü 'temp' değişkenine atanır ve bir *while* döngüsü ile listenin son düğümüne kadar ilerlenir (temp listenin son düğümünü gösterir). Sonra yeni düğüm 'temp' değişkeninin bir sonraki düğümü gösteren 'ptr' işaretçisine atanır. Tabii ki yeni düğümün 'value' değişkenine kullanıcıdan okunan değer, 'ptr' de-

ğişkenine de NULL değeri atanmış durumdadır. Bu fonksiyonda dikkat edilmesi gereken durum herhangi bir işaretçinin çevrilmemiş olmasıdır. Bağlı listenin başına yeni bir düğüm eklediğimizde 'head' işaretçisi çevrilmişti. Çünkü listenin ilk düğümü değişmişti. Ancak bağlı listenin sonuna ekleme yapıldığında 'head' işaretçisinde bir değişiklik olmadı ve bu değişken çevrilmedi.

Şekil 3.8'deki *deleteNode* fonksiyonu bir bağlı listenin ilk düğümünü silmek için yazılmıştır.

```

1 struct node *deleteNode (struct node *head)
2 {
3     struct node *temp;
4     temp=head;
5     head=head->ptr;
6     delete temp;
7     return head;
8 }

```

Şekil 3.8. Bağlı listenin ilk düğümünü silme fonksiyonu

Yukarıda verilen fonksiyonda bağlı listenin silinecek ilk düğümü geçici bir işaretçiye atanmıştır. Sonra ilk düğüm (head) bağlı listenin ikinci düğümü olacak şekilde değiştirilmiştir. Geçici işaretinin gösterdiği alan (eski ilk düğüm) *delete*

fonksiyonu ile belleğe geri verilmiştir. Bağlı listenin ilk düğümü değiştiği için bu değer çevrilmiştir.

Şekil 3.9'daki *deleteEnd* fonksiyonu bir bağlı listenin son düğümünü silmek için yazılmıştır.

```

1 void deleteEnd (struct node *head)
2 {
3     struct node *temp1, *temp2;
4     temp1=head;
5     temp2=head;
6
7     while(temp1->ptr!=NULL)
8     {
9         temp2 =temp1;
10        temp1=temp1->ptr;
11    }
12
13    temp2->ptr=NULL
14    delete temp1;
15 }

```

Şekil 3.9. Bağlı listenin en son düğümünü silme fonksiyonu

Son düğümü silmek için yukarıda yer alan fonksiyonda iki farklı geçici işaretçi kullanılmıştır. *while* döngüsü ile iki işaretçi bağlı liste üzerinde arka arkaya hareket ettirilmiş ve döngü sonlandığında işaretçilerin son iki düğümü göstermesi sağlanmıştır. Son düğümü gösteren işaretçi, bu alanı belleğe geriye vermek için *delete* fonk-

siyonu ile silinmiştir. Bir önceki düğümün 'ptr' işaretçi değişkeni NULL değerini almış ve listeyi sonlandırmıştır. Bu şekilde bağlı listenin son düğümü silindiğinde, listenin ilk düğümünün adresi değişmemiş bu nedenle fonksiyondan herhangi bir değer çevrilmemiştir.



ARAŞTIRALIM

Bağlı listelerin başı ve sonu dışında herhangi bir noktasına yeni bir düğüm eklemeyi veya ilk ve son düğümleri dışındaki herhangi bir düğümü silmeyi araştıralım.

Bağlı listeler üzerinde arama işlemi ilk düğümünden son düğüme kadar gezinerek sıralı bir şekilde gerçekleştirilir (Şekil 3.10). Aranılan değer

bulunduğu zaman ilgili düğümün adresi çevrilir. Eğer aranan değer bulunamaz ise NULL işaretçi değeri döndürülür.

```

1 struct node *find (struct node *head, int key)
2 {
3     struct node *temp;
4     temp=head;
5     if (temp->value == key)
6         return temp;
7
8     while(temp!=NULL)
9     {
10        temp=temp->ptr;
11        if(temp->value==key)
12            return temp;
13    }
14    return NULL;
15 }

```

Şekil 3.10. Bağlı listede verilen bir değeri arama fonksiyonu

2.3. Dairesel Bağlı Listeler

Bağlı listelerde, listenin sonundaki düğümde yer alan ve bir sonraki düğümü gösteren işaretçi NULL değerini alır. Dairesel bağlı listelerde ise

listenin sonundaki düğüm ilk düğümü gösterir (Şekil 3.11).



Şekil 3.11. Dairesel bağlı liste

Dairesel bağlı listeler üzerinde işlem yaparken dikkat edilmesi gerekli durum listenin sonuna ulaşıp ulaşılmadığının kontrol edilmesidir. Bu kontrol işleminde *struct* yapısında yer alan 'ptr' işaretçisinin gösterdiği değer NULL değeri ile değil, ilk düğümün adresi ile karşılaştırılır.

Şekil 3.12'de yer alan *display* fonksiyonu bir dairesel bağlı listedeki düğümlerin değerlerini ekrana yazdırmak için oluşturulmuştur.

```
1 void display (struct node *head)
2 {
3     struct node *temp;
4     temp=head;
5
6     cout <<temp->value<<' ';
7     while (temp->ptr!=head)
8     {
9         temp=temp->ptr;
10        cout <<temp->value<<' ';
11    }
12    cout<<endl;
13 }
```

Şekil 3.12. Dairesel bağlı liste yazdırma fonksiyonu

2.4. Çift Yönlü Bağlı Listeler

Bir önceki kısımda çalıştığımız bağlı listelerde bir düğüm sadece kendisinden bir sonraki düğümü göstermekteydi. Bir düğümün aynı zamanda kendisinden önce ve kendisinden sonra-

ki düğümleri gösterdiği bağlı listelere çift yönlü bağlı listeler denir. Çift yönlü bağlı listelerdeki düğümlerde kullanılan *struct* yapısı Şekil 3.13'te verilmiştir.

```
1 struct nodeDouble
2 {
3     int value;
4     struct node *ptrNext;
5     struct node *ptrPrev;
6 }
```

Şekil 3.13. Çift yönlü bağlı liste düğümü için struct tanımı



Şekil 3.14. Çift yönlü bağlı liste

Çift yönlü bağlı listelerde yer alan düğümler iki adet işaretçiye sahiptir. İlk işaretçi bir sonraki

düğümü, ikinci işaretçi bir önceki düğümü göstermektedir (Şekil 3.14). Çift yönlü bağlı listedeki

ilk düğümün 'ptrPrev' işaretçisi NULL değerini gösterir. Aynı şekilde çift yönlü bağlı listedeki son düğümün 'ptrNext' işaretçisi NULL değerine sahiptir.

Şekil 3.15'te verilen *insert* fonksiyonu çift yönlü bağlı listenin başına yeni bir düğüm yerleştirir.

```

1  struct nodeDouble *insert(struct nodeDouble * head)
2  {
3      struct nodeDouble *newNode;
4      newNode=new struct nodeDouble;
5      int newValue;
6
7      cin>>newValue;
8      newNode->value=newValue;
9      newNode->ptrNext=head;
10     newNode->ptrPrev=NULL;
11     head->ptrPrev=newNode;
12     head=newNode;
13
14     return head
15 }

```

Şekil 3.15. Çift yönlü bağlı listenin başına düğüm ekleme fonksiyonu

Fonksiyon öncelikle 'newNode' isminde yeni bir çift yönlü bağlı liste düğümü oluşturur. Bu yeni düğüm için gerekli alan *new* fonksiyonu ile ayrılır. Kullanıcıdan 'newValue' isminde yeni bir tamsayı okunur ve yeni düğümün 'value' değişkenine atanır. Yeni düğümün 'ptrNext' işaretçisi önceki 'head' düğümünü gösterir. Önceki 'head' düğümün bir önceki düğümü gösteren 'ptrPrev'

işaretçisi, yeni eklenen 'newNode' düğümünü gösterir. Yeni eklenen düğüm bağlı listenin yeni 'head' düğümü olur ve fonksiyon bu değeri çevirir. Fonksiyonun dönüş tipi bir *struct nodeDouble* işaretçisidir.

Şekil 3.16'daki *insertEnd* fonksiyonu çift yönlü bağlı listenin sonuna yeni bir düğüm eklemektedir.

```

1  void insertEnd (struct nodeDouble *head)
2  {
3      struct nodeDouble *newNode;
4      struct nodeDouble *temp
5      int newValue;
6
7      temp=head;
8      newNode=new struct nodeDouble;
9      cin>>newValue;
10     newNode->value=newValue;
11     newNode->ptrNext=NULL;
12
13     while (temp->ptrNext!=NULL)
14         temp=temp->ptrNext;
15
16     temp->ptrNext=newNode;
17     newNode->ptrPrev=temp;
18 }

```

Şekil 3.16. Çift yönlü bağlı listenin sonuna düğüm ekleme fonksiyonu

Öncelikli olarak yeni bir düğüm oluşturulur. Düğümün 'value' değeri olarak kullanıcıdan alınan değer atanır. Çift yönlü bağlı listenin sonuna bir düğüm eklemek için, listenin sonuna kadar ilerlemek gerekir. Listenin son düğümüne ulaştığımızda, bu düğümün 'ptrNext' işaretçisi kullanılarak listenin sonuna yeni düğüm eklenir.

Yeni düğümün bir önceki düğümü olarak listenin önceki son düğümü atanır. Çift yönlü bağlı listenin ilk düğümü değişmediği için herhangi bir değer bu fonksiyon tarafından çevrilmemektedir. Fonksiyonun dönüş tipi *void* olarak belirlenmiştir.

Şekil 3.17'deki *deleteNode* fonksiyonu bir çift yönlü bağlı listenin ilk düğümünü silmek için kullanılır.

```

1 struct nodeDouble *deleteNode (struct nodeDouble *head)
2 {
3     struct nodeDouble *temp;
4     temp=head;
5     head=head->ptrNext;
6     head->ptrPrev=NULL;
7     delete temp;
8     return head;
9 }
```

Şekil 3.17. Çift yönlü bağlı listenin ilk düğümünü silme fonksiyonu

Bu fonksiyonda çift yönlü bağlı listenin silinecek ilk düğümü geçici bir işaretçiye atanmıştır. Sonra ilk düğüm (head) çift yönlü bağlı listenin ikinci düğümü olacak şekilde değiştirilmiştir. Yeni ilk düğümün 'ptrPrev' işaretçisine NULL değeri verilmiştir. Geçici işaretinin gösterdiği alan (eski ilk düğüm) *delete* fonksiyonu ile belleğe

geri verilmiştir. Çift yönlü bağlı listenin ilk düğümü değiştiği için bu değer fonksiyon tarafından çevrilmiştir.

Şekil 3.18'deki *deleteEnd* fonksiyonu bir çift yönlü bağlı listenin son düğümünü silmek için yazılmıştır.

```

1 void deleteEnd(struct nodeDouble *head)
2 {
3     struct nodeDouble *temp;
4     temp=head;
5
6     while(temp->ptrNext!=NULL)
7         temp=temp->ptrNext;
8
9     temp->ptrPrev->ptrNext=NULL;
10    delete temp;
11 }
```

Şekil 3.18. Çift yönlü bağlı listenin son düğümünü silme fonksiyonu

Bu fonksiyonda, bir geçici işaretçi (temp) kullanılarak çift yönlü bağlı listenin en son elemanı silinmiştir. Bir *while* döngüsü içinde 'temp' işaretçisi dizinin sonuna kadar hareket eder. Döngü sonlandığında 'temp' işaretçisi en son düğümü gösterir. Bir önceki düğümü gösteren 'ptrPrev'

işaretçisi kullanılarak bir önceki düğümüne erişilir ve bu düğümün 'ptrNext' işaretçi değeri NULL yapılır. Son olarak en son düğümü gösteren 'temp' işaretçisi *delete* ile silinir ve ilgili düğümün kapladığı alan belleğe geri verilir.

BÖLÜM ÖZETİ

Bu bölümde öncelikli olarak veri yapıları kavramı ve önemi anlatılmıştır. Temel veri yapıları işlemleri sıralanmıştır. Bir veri yapısı üzerinde gerçekleşmesi beklenen temel işlemler gezinme, ekleme, silme, arama ve birleştirme olarak ifade edilebilir. İlk olarak anlatılan veri yapısı dizilerdir. Dizilerin temel özellikleri çalışılmış, bir dizi değişkeninin nasıl tanımlanacağı, dizi elemanlarına ne şekilde değer atanacağı gösterilmiştir. Bir dizi üzerinde gezinme işlemi, diziye yeni eleman ekleme, diziden bir eleman silme ve diziler üzerinde arama işlemi anlatılmıştır. Dizi elemanları bellekte birbirini takip eden alanlarda yer alırlar. Ayrıca diziler tanımlandığında boyutlarının belirlenmiş olması gerekir. Dizilerden sonra çalışılan ve dizilerin yukarıda ifade edilen iki dezavantajına çözüm getiren veri yapısı, bağlı listelerdir. Bağlı listelerin boyutu liste ilk oluşturulduğu zaman belirlenmek zorunda değildir. Yeni bir eleman listeye eklendiğinde dinamik olarak bellekten yer ayrılır. Ayrıca bağlı listelerin elemanlarının bellekte ardışık alanlarda yer alma zorunluluğu bulunmamaktadır. Bağlı listeler, tek yönlü bağlı listeler, dairesel bağlı listeler ve çift yönlü bağlı listeler olarak farklı gruplara ayrılabilir. Tek yönlü bağlı listelerde her bir düğüm listede kendisinden sonra gelen düğümü gösterir. Dairesel bağlı listelerde listenin son düğümünden listenin ilk düğümüne bağlantı vardır. Çift yönlü bağlı listelerde her bir düğüm iki işaretçi içerir. Bu işaretçiler yardımı ile düğümler kendilerinden önceki ve sonraki düğümleri gösterirler.

GÖZDEN GEÇİRELİM

1. Aşağıdakilerden hangisi temel veri yapıları işlemlerinden birisi değildir?

- a) Gezinme
- b) Ekleme
- c) Arama
- d) Silme
- e) Çarpma

2. Aşağıdaki dizi tanımlamalarından hangisi yanlıştır?

- a) `int a[5]={0};`
- b) `int a[4];`
- c) `int a[4]={1, 2, 3, 4, 5};`
- d) `int a[5]={1, 2, 3};`
- e) `int a[]={1, 2, 3};`

3. `int a[5]={1, 2, 3, 4, 5};`

tanımlaması verilmiş olsun. Aşağıdakilerden hangisi dizinin ilk elemanına erişmek için kullanılamaz?

- a) `a[0]`
- b) `*a`
- c) `*(&a[0])`
- d) `a[1]`
- e) `a[4-3-1]`

4. `int b[5]={0, 2, 4, 6, 8};`

tanımlaması verilmiş olsun. Aşağıdakilerden hangisi dizinin tüm elemanlarını doğru bir şekilde ekrana yazdırır?

- a) `for(int i=0; i<5; i++)`
`cout<<b[i]<<' '`;
- b) `for(int i=1; i<5; i++)`
`cout<<b[i]<<' '`;
- c) `for(int i=0; i<=5; i++)`
`cout<<b[i]<<' '`;
- d) `for(int i=0; i<4; i++)`
`cout<<b[i]<<' '`;
- e) `for(int i=1; i<=5; i++)`
`cout<<b[i]<<' '`;

5. `int c[5]={1, 3, 5, 7, 9};`

tanımlaması verilmiş olsun. Aşağıdakilerden hangisi dizinin 3. elemanına değer atamak için kullanılamaz?

- a) `cin>>a[2];`
- b) `a[2]=15;`
- c) `20=a[2];`
- d) `a[2]=a[2]+5;`
- e) `a[2]++;`

6. Aşağıdakilerin hangisi bağlı listelerin dizilere göre üstün olduğu durumlardan biridir?

- a) Doğrudan elemanlara erişim
- b) Boyutunun önceden belirlenme zorunluluğu olmaması
- c) Kolay arama işlemi
- d) Kolay kodlama işlemi
- e) İşlemlerin daha hızlı yapılabilmesi

7. Aşağıdakilerin hangisi dizilerin bağlı listelere göre üstün olduğu durumlardan biridir?

- a) Boyutunun önceden belirlenme zorunluluğu
- b) Bellek yönetimi
- c) Sabit belirlenmiş boyut
- d) Doğrudan elemanlara erişim
- e) Kolay arama işlemi

9. struct node{

int v;

struct node *p;

}*node1;

Yukarıdaki düğüm yapısı verildiğinde, aşağıdakilerin hangisi tek yönlü bağlı listede, listenin son düğümünde olup olmadığımızı kontrol için kullanılır?

- a) if(node1.p==NULL)
- b) if(node1->p==NULL)
- c) if(node1==NULL)
- d) if(p==NULL)
- e) if(node1==p)

8. Aşağıdakilerin hangisi basit çift yönlü bağlı listenin özelliklerinden birisi değildir?

- a) Her düğümde iki işaretçiye sahip olması
- b) Liste üzerinde iki yönde gezinmeye imkan vermesi
- c) Tek yönlü bağlı listeye göre fazla bellek kullanımı olması
- d) Bir listede en az iki NULL işaretçi olması
- e) Listenin sabit boyutlu olması

10. struct node{

int v;

struct node *p;

}*node1;

Yukarıdaki düğüm yapısı verildiğinde, aşağıdakilerin hangisi tek yönlü bağlı listede, bir sonraki düğüme ilerlemek amacıyla kullanılır?

- a) node1=node1->p;
- b) node1=p;
- c) node1=node1.p;
- d) node1.p=NULL;
- e) node1=node1+1;

Yanıt Anahtarı: 1-E 2-C 3-D 4-A 5-C 6-B 7-D 8-E 9-B 10-A

Kaynakça

Deitel, P., Deitel, H. (2017). C++ How to Program. 10th Edition. Pearson.

Thareja R. (2014) Data Structures Using C, 2nd Edition, Oxford University Press.

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 4

KUYRUKLAR VE YIĞINLAR

ANAHTAR KAVRAMLAR

Kuyruk, Yığın, Enqueue, Dequeue, Push, Pop

ÖĞRENME HEDEFLERİ

- 1 Kuyruk veri yapısını tanımlayabilir ve kullanabilir.
- 2 Kuyruk veri yapısı üzerindeki temel işlemleri uygulayabilir.
- 3 Yığın veri yapısını tanımlayabilir ve kullanabilir.
- 4 Yığın veri yapısı üzerindeki temel işlemleri uygulayabilir.
- 5 Yığın veri yapısını gerçek problemlerin çözümünde kullanabilir.

GİRİŞ

Bu bölümde kuyruk ve yığın veri yapıları anlatılacaktır. Kuyruk bir listeye ilk giren

liste elemanının ilk çıkması, bir başka ifade ile ilk işleme alınması kuralını uygulayan bir veri yapısıdır. Bu nedenle ilk giren ilk çıkar (First In First Out - FIFO) tipi bir yapı olarak değerlendirilir. Bir marketin kasasında bekleyen alışveriş yapan kişiler, bir bankada işlem bekleyen müşteriler günlük hayattan kuyruk veri yapısına uygun örneklerdir. Bu yapıyı uygulamak için diziler veya bağlı listeler kullanılabilir. Her iki yapıda da kuyruk veri yapısına ait temel işlemler değişmez. Sadece diziler veya bağlı listelerin karakterine göre kodlama işlemi farklılık gösterir. Kuyruk veri yapısında listenin her iki ucu da aktif olarak kullanılır. Kuyruk veri yapısında listenin sonu dışında herhangi bir noktaya ekleme yapılamaz. Aynı şekilde listenin başı dışında herhangi bir yerinden de silme işlemi gerçekleştirilemez. Kullanılan yapıya göre listenin sonu veya başı dışındaki bir elemana doğrudan erişmek, kullanılan programlama dilinin sözdizimine göre mümkün olabilir. Ancak bu kuyruk veri yapısında bu işlemin geçerli bir işlem olduğu anlamına gelmez. Eğer bu tür bir rasgele değişikliğe (ekleme veya silme) kodlamada izin verilmiş ve uygulanmışsa bu veri yapısı kuyruk olarak değerlendirilemez. Yığınlar listeye son giren elemanın ilk çıkması kuralına sahip veri yapılarıdır. Son giren ilk çıkar (Last In First Out - LIFO) tipi yapılar olarak bilinirler. Temizlenen tabakların üst üste yerleştirilmesi yığınlar için örnek olarak verilebilir. Temiz bir tabağa ihtiyaç duyulduğunda en son yıkanan bir başka ifade ile tabak yığınının en üstünde yer alan tabak kullanılır. Kuyruklarda olduğu gibi yığınları uygulamak için diziler veya bağlı listeler kullanılabilir. Her iki yapının birbirlerine göre avantaj ve dezavantajları mevcuttur. Kuyruk veri yapısının aksine yığınlar listenin sadece bir ucu kullanılır. Ekleme ve silme işlemleri aynı noktadan gerçekleştirilir. Ekleme ve silme işlemleri kesinlikle listenin farklı bir noktasına uygulanamaz. Bu işlemlerinin listede rasgele herhangi bir noktada gerçekleştiği yapılar yığın olarak isimlendirilemezler.



Bu bölümde önce kuyruk veri yapısı anlatılacak, bu yapının temel fonksiyonları gösterilecektir. Daha sonra yığın veri yapısı ve bu yapı üzerinde

gerçekleştirilebilecek işlemler çalışılacaktır. Bölüm, yığın veri yapısının pratik olarak kullanıldığı bir örnek ile tamamlanacaktır.

1. KUYRUKLAR

Günlük yaşamda birçok yerde karşılaşılabileceğimiz kuyruk kavramı temel veri yapılarından birisidir. Bir otobüs durağında bekleyen yolcular, markette kasada bekleyen müşteriler ve yemekhanede yemek sırasında yer alan öğrenciler kuyruk veri yapısına verilebilecek örneklerdendir. Doğrusal bir veri yapısı olup ilk giren ilk çıkar prensibi ile çalışır. Listenin bir tarafı yeni bir

eleman eklemek için belirlenmiş olup diğer taraf silme veya kuyruktan çıkarma işlemi için kullanılır.

Kuyruk veri yapısını uygulamak için diziler veya bağlı listeler kullanılabilir. Şekil 4.1'de verilen dizi ile dört elemana sahip örnek bir kuyruk gösterilmiştir. Bu kuyruk belirli ekleme ve silme işlemlerinden sonra bu duruma gelmiştir.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
		14	-42	50	61				

Şekil 4.1 Dizi kullanarak örnek bir kuyruk uygulaması

Burada kuyruk veri yapısını basit bir şekilde ifade edebilmek amacıyla tamsayı kullanılmıştır. Gerçek uygulamalarda her türlü nesneye ait kuyruk oluşturmak mümkündür. Buradaki tamsayıları, bu tür nesneleri temsil eden anahtar değerler olarak da görebiliriz. Kuyruğa yeni bir eleman ekleneceği zaman, bu elemanın yerleştirileceği yer bu dizide a[6] pozisyonudur. Yeni bir elemanın başka bir indise yerleştirilmesi mümkün değildir. Aynı şekilde kuyruktan ilk alınacak/silinecek eleman da kesinlikle bellidir. Yukarıda verilen kuyruktan silinecek ilk eleman 14 değerine sahip a[2] pozisyonudur. Farklı bir eleman

çıkartılamaz. Eğer kuyrukta herhangi bir eleman yok ise, kuyruk boşsa silme işlemi gerçekleştirilemez. Benzer şekilde dolu olan bir kuyruğa yeni bir eleman eklenemez. Yukarıdaki kuyruk maksimum on eleman alabilir. Son pozisyon olan a[9] alanı kullanıldıktan sonra, a[0] pozisyonu ile ekleme işlemine devam edilmelidir. İlgili fonksiyonlar kodlanırken bu durumlar göz önüne alınmalıdır. Ekleme ve silme işlemleri gerçekleştirilirken kuyruk veri yapısı için iki değişken kullanılır. Bu değişkenlerden birincisi kuyruktaki son elemanı gösteren 'rear' indisidir. Diğeri ise kuyruktaki ilk elemanı gösteren 'front' indisidir.

```

1 void enqueue (int queue[], int value)
2 {
3     if((rear+1)%SIZE != front)
4     {
5         rear=(rear+1)%SIZE;
6         queue[rear]= value;
7     }
8     else
9         cout<<"Kuyruk dolu. Yeni eleman eklenemez. \n";
10 }

```

Şekil 4.2 Bir kuyruğa yeni bir eleman ekleme fonksiyonu



DİKKAT EDELİM

enqueue ve **dequeue** fonksiyonlarında kullanılan 'front' ve 'rear' değişkenlerinin ilgili programda global değişken olarak tanımlandığını varsayalım.

Şekil 4.2 ve 4.3'te verilen fonksiyonlarda SIZE ifadesinin daha önce tanımlanmış sembolik sabit, 'rear' ve 'front' değişkenlerinin ise global değişken olduğunu kabul edelim. Şekil 4.2' de dizi ile gösterilen bir kuyruk veri yapısına yeni bir eleman ekleyen *enqueue* fonksiyonunun tanımı verilmiştir. Bu fonksiyonda önce son elemanı gösteren 'rear' değişkeni bir artırılmış ve bu değişkenin gösterdiği alana yeni değer atanmıştır. Bu fonksiyon çağrıldığında kuyruksa en az bir eleman olduğu varsayılmıştır. Şekil 4.3'te ise kuy-

ruktan bir eleman çıkarma fonksiyonu *dequeue* tanımlanmıştır. Önce 'front' indisinin gösterdiği değer geçici bir değişkene atanmış, 'front' değeri bir artırılmıştır. Fonksiyonun sonunda geçici değişkene atanan değer döndürülmüştür. Fonksiyonlarda 'rear' ve 'front' indislerinin SIZE-1 den sonra 0 ile devam etmesi '%' işleci ile sağlanmıştır. Ayrıca dolu olan bir kuyruğa yeni eleman eklenemeyeceği, boş olan bir kuyruktan eleman silinemeyeceği uyarıları kullanıcıya aktarılmıştır.

```

1  int dequeue (int queue [])
2  {
3      int temp=-1;
4      if (front==-1 && rear==-1)
5      {
6          cout<<"Kuyruk bos. Eleman silinemez\n";
7          return temp;
8      }
9      else
10     {
11         temp=queue[front];
12         front=(front+1)%SIZE;
13         return temp;
14     }
15 }

```

Şekil 4.3 Bir kuyruktan bir eleman çıkarma fonksiyonu

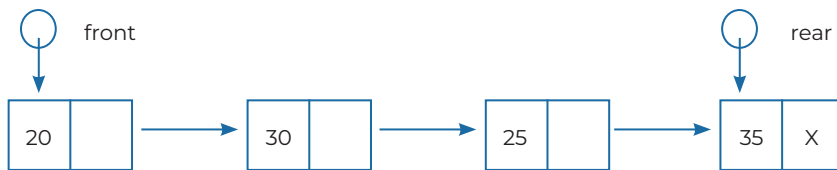


ARAŞTIRALIM

Kuyruğa yeni eklenen elemanların belirli özelliklerine göre farklı noktalarda yer bulabilmesine olanak sağlayan öncelikli kuyruk veri yapısını araştıralım.

Kuyrukları uygulamak için aynı zamanda bağlı listeler de kullanılabilir. Bir bağlı listede listenin ilk ve son düğümlerini gösteren işaretçiler kuyruk veri yapısını uygulamak için gereklidir. Bağlı listenin ilk düğümü listeden alınacak/sili-

necek ilk elemanı gösterir. Listenin en son düğümünü gösteren işaretçi yardımıyla da listeye yeni bir eleman eklenebilir. Şekil 4.4'te dört elemana sahip bir kuyruğun tek yönlü bağlı liste ile gösterimi verilmektedir.



Şekil 4.4 Dört elemandan oluşan bir kuyruk veri yapısının bağlı liste ile gösterimi

Bağlı listeler kullanılarak bir kuyruk oluşturulduğunda *struct* yapısı kullanarak kuyruk tanımlaması yapılabilir (Şekil 4.5). *struct queue* yapısı ile kuyruk veri tipi tanımlanmıştır. Bağlı liste

ile kuyruk oluşturulduğunda, listedeki ilk ve son düğümün adresi bu tanımlama için yeterlidir. Bu işaretçiler *struct node* tipinde tanımlanmış ve ‘front’ ile ‘rear’ olarak isimlendirilmişlerdir.

```
1 struct queue
2 {
3     struct node *front;
4     struct node *rear;
5 };
```

Şekil 4.5 Bir kuyruk veri yapısı için *struct* yapısı

Kuyruğu oluşturan her bir düğüm farklı bir *struct* ile oluşturulur (Şekil 4.6). *struct node* yapısı bağlı listede yer alan düğümleri oluşturmak için tanımlanmıştır. Her düğümün veri olarak sadece

tamsayı değere sahip olduğu varsayılmıştır. Bağlı listelerdeki düğümler bir sonraki düğümü göstermek için kendi tiplerinden bir işaretçiye sahip olurlar.

```
1 struct node
2 {
3     int value;
4     struct node *ptr;
5 };
```

Şekil 4.6 Kuyruk veri yapısında kullanılan düğümler için *struct* yapısı

Şekil 4.7 ve 4.8’de sırasıyla bağlı liste kullanılarak kuyruk uygulamasında kuyruğa yeni bir eleman ekleme ve kuyruktan bir eleman silme için kullanılan fonksiyonlar verilmiştir.

```
1 void enqueue (struct queue *k, int number)
2 {
3     struct node *newNode=new struct node;
4     newNode->value=number;
5     newNode->ptr=NULL;
6     k->rear->ptr=newNode;
7     k->rear=newNode;
8 }
```

Şekil 4.7 Bağlı liste ile gösterilen bir kuyruğa yeni bir eleman ekleme fonksiyonu

```
1 void dequeue (struct queue *k)
2 {
3     struct node *temp;
4     temp=k->front;
5     k->front=k->front->ptr;
6     delete temp;
7 }
```

Şekil 4.8 Bağlı liste ile gösterilen bir kuyruktan bir eleman çıkarma fonksiyonu

enqueue fonksiyonunda *new* fonksiyonu ile yeni bir düğüm oluşturulmuş, bu düğümün “value” değişkenine, eklenecek tamsayı atanmıştır. Yeni düğümün bir sonraki düğümü gösteren işaretçisine ise NULL değeri atanmıştır. ‘k’ kuyruk değişkeninin ‘rear’ düğümünün işaretçisine yeni düğüm atanmış, bu şekilde yeni düğüm kuyruğun sonuna eklenmiştir. *enqueue* fonksiyonunun tanımlanmasında bellekte yeni düğüm için yer ayrılmasında sorun olmayacağı kabul edilmiştir.



ARAŞTIRALIM

Bağlı liste ile kuyruk uygulamasında **enqueue** fonksiyonunda yeni düğüm için bellekte alan ayrılmama durumunu fonksiyon içinde kontrol etmeyi araştırınız.



Ayrıca bu fonksiyon çağrıldığında kuyrukta en az bir eleman olduğu varsayılmıştır.

dequeue fonksiyonunda kuyruk yapısının ilk düğümü ('front' düğümü) geçici bir işaretçiye atanmıştır. İlk düğüm, kuyruktaki ikinci düğümü gösterecek şekilde değiştirilmiştir. Geçici bir işaretçiye atanan düğüm ise *delete* fonksiyonu ile silinmiştir. *dequeue* fonksiyonunda kuyrukta eleman olduğu kabul edilmiştir.



ARAŞTIRALIM

Bağlı liste ile kuyruk uygulamasında **dequeue** fonksiyonunda kuyrukta eleman olup olmadığını fonksiyon içinde kontrol etmeyi araştıralım.

2. YIĞINLAR

Bir listeye en son eklenen elemanın ilk kullanılan/silinen eleman olması durumunu uygulamak için yığın veri yapısı kullanılır. Bu veri yapısı son giren ilk çıkar prensibi ile çalışan bir yapı olarak değerlendirilir. Bir programın çalışması sırasında çağrılan her fonksiyonun yeni bir fonksiyon çağırma durumu yığın veri yapısı için iyi bir örnektir. Tüm açık fonksiyonlar bir yığına eklenmiş olarak düşünülebilir. En son çağrılan fonksiyon işlemini tamamladığında yığından silinir. İşlemini tamamlama sırası, en son fonksiyonu çağırarak fonksiyondadır. Bu fonksiyonda işlemi-

ni tamamlayarak yığından uzaklaştırılır. Fonksiyonların tamamlanma ve yığından alınma süreci bu şekilde devam eder. Yığına en son eklenen fonksiyon, yığından ilk silinen fonksiyon olacak şekilde, işlem tamamlanır. Buradaki örnekte de görüldüğü gibi yığın veri yapısında listenin sadece bir ucu kullanılır. Ekleme ve silme işlemleri aynı noktadan gerçekleştirilir.

Kuyrukları olduğu gibi yığın veri yapısını uygulamak için diziler ve bağlı listeler kullanılabilir. Şekil 4.9'da verilen dizi ile bir yığını ifade edebiliriz.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
57	43	14	-42	50	61				

Şekil 4.9 Dizi kullanılarak örnek bir yığın uygulaması

Bir dizi ile yığını ifade ettiğimizde tamsayı bir indis değeri bize yığının en üstteki değerini gösterir. Bu değere 'top' ismi verilir. Yukarıdaki örnekte 'top' değeri 5 olarak görülmektedir. Bu yığına yeni bir tamsayı eklemek isteyelim. Bu

durumda 'top' değeri 1 artırılabilecek ve 6 olacaktır. Yeni değer de bu alana yazılacaktır. Eğer 75 değeri yığına eklenirse Şekil 4.10'da verilen dizi oluşur.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
57	43	14	-42	50	61	75			

Şekil 4.10 Yığın veri yapısına yeni bir eleman eklenmesi (75 eklenmiştir)

Silme işlemi de aynı noktadan gerçekleştirilir. Arka arkaya üç kez silme işlemi uyguladığımızı

düşünelim. Bu durumda yeni liste Şekil 4.11'deki gibi olacaktır. Yeni 'top' değeri ise 3'tür.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
57	43	14	-42						

Şekil 4.11 Yığın veri yapısından arka arkaya üç eleman silinmesi

Diziler kullanılarak yığın veri yapısı uygulandığında, yığına yeni bir eleman ekleme işlemini

gerçekleştiren *push* fonksiyonu Şekil 4.12’de verilmiştir.

```

1 void push (int stack[], int value)
2 {
3     if (top+1<SIZE-1)
4     {
5         top++;
6         stack[top]=value;
7     }
8     else
9         cout<<"Yigin dolu. Yeni eleman eklenemez.\n";
10 }

```

Şekil 4.12 Yığına yeni bir eleman ekleme fonksiyonu

Şekil 4.13’te diziler kullanılarak yığın veri yapısı uygulandığında, yığından bir eleman silme

işlemini gerçekleştiren *pop* fonksiyonu verilmiştir.

```

1 int pop (int stack [])
2 {
3     int temp=-1;
4     if (top!=-1)
5     {
6         temp=stack[top];
7         top--;
8         return temp;
9     }
10    else
11    {
12        cout<<"Yigin bos. Eleman silinemez.\n";
13        return temp;
14    }
15 }

```

Şekil 4.13 Yığından bir eleman çıkarma fonksiyonu

push fonksiyonunda öncelikli olarak ilgili yığının boş alana sahip olup olmadığı kontrol edilir. Yığında boş alan mevcutsa ‘top’ değişkeni 1 arttırılır. ‘top’ indisinin gösterdiği alana eklenmek istenen değer yazılır. Eğer yığında boş alan yok ise kullanıcıya yığının dolu olduğu ve yeni eleman ekleyemeyeceği bilgisi verilir.

pop fonksiyonunda yığında eleman olup olmadığı kontrol edilir. Eğer yığında eleman var ise ‘top’ değişkeninin gösterdiği eleman geçici

bir değişkene atanarak çevrilir. ‘top’ değişkeni bir azaltılır. Yığında eleman yoksa bu durum kullanıcıya bilgi olarak verilir.

Bağlı listeler kullanılarak da yığınlar oluşturulabilir. Bağlı listenin sadece bir ucu kullanılır. Ekleme ve silme işlemleri aynı uçtan gerçekleştirilir. Bu uçtaki ilk düğümü gösteren işaretçi değişkeni ‘top’ olarak isimlendirilir. Yığınları bağlı listeler ile ifade ederken Şekil 4.14’te verilen *struct* yapısı ile oluşturulan düğümler kullanılır.

```

1 struct node
2 {
3     int value;
4     struct node *ptr;
5 };

```

Şekil 4.14 Yığın veri yapısında kullanılan düğümler için struct yapısı

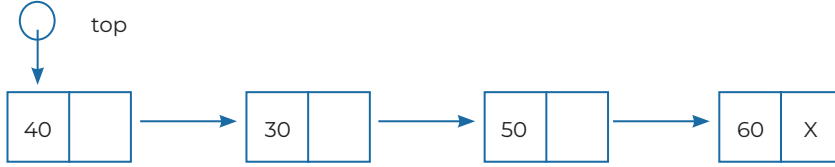
Bir yığın bağlı liste kullanılarak oluşturulduğunda, bu yığına erişmek için sadece bir *struct node* işaretçisi yeterlidir. Şekil 4.15’te verilen yı-

ğında ‘top’ işaretçisi tamsayı değeri 40 olan düğümü göstermektedir.



DİKKAT EDELİM

Bir yığında ekleme ve silme işlemleri listenin aynı noktasından gerçekleşmektedir.



Şekil 4.15 Dört elemandan oluşan bir yığın veri yapısının bağlı liste ile gösterimi

Bu yığına yeni bir eleman eklendiğinde 'top' işaretçisi bu yeni elemanı, yeni eleman da yığının eski 'top' düğümünü gösterecektir. Silme işlemi de aynı uçtan gerçekleşir. Eğer yukarıda verilen yığından arka arkaya 2 eleman silinirse, sırasıyla

40 ve 30 değerlerine sahip düğümler uzaklaştırılacak ve 'top' işaretçisi 50 tamsayı değerine sahip düğümü gösterecektir. Şekil 4.16 ve 4.17'de sırasıyla bağlı liste ile yığın uygulamasında kullanılan *push* ve *pop* fonksiyonları verilmektedir.

```

1 struct node *push(struct node *t, int number)
2 {
3     struct node *newNode=new struct node;
4     newNode->value=number;
5     newNode->ptr=t;
6     t=newNode;
7     return t;
8 }
  
```

Şekil 4.16 Bağlı liste ile gösterilen bir yığına yeni bir eleman ekleme fonksiyonu

```

1 struct node *pop(struct node *t)
2 {
3     struct node *temp;
4     temp=t;
5     t=t->ptr;
6     delete temp;
7     return t;
8 }
  
```

Şekil 4.17 Bağlı liste ile gösterilen bir yığından bir eleman çıkarma fonksiyonu

Bir yığını temsil etmek için sadece bir düğüm işaretçisi gerektiğinden yığın oluşturmak için ayrı bir *struct* yapısına ihtiyaç duyulmamıştır. *push* fonksiyonunda yeni bir düğüm oluşturulmuş, eklenecek tamsayı bu düğümün değeri olarak atanmıştır. Yeni düğümün işaretçi değeri eski 'top' elemanı gösterecek şekilde atanmıştır. 'top' düğüm is yeni eklenen düğüm olarak değiştirilmiştir. Bu düğümü gösteren işaretçi fonksi-

yondan çevrilmiştir. Ayrıca *push* fonksiyonunda yeni oluşturulan düğüm için bellekte yer ayırma işleminin başarılı olduğu varsayılmıştır. *pop* fonksiyonu eski 'top' düğümü geçici bir işaretçiye atamış ve 'top' düğümü değiştirmiştir. Geçici işaretçiye atanan düğüm ise *delete* ile silinerek, ilgili alan belleğe geri verilmiştir. *pop* fonksiyonunda, yığında eleman olduğu varsayılmıştır.

3. UYGULAMA

Yığınları kullanarak bir karakter dizisini tersten yazdırma işlemi bu bölümde uygulama olarak gösterilecektir. Şekil 4.18’de verilen programda, önce karakter dizisi kullanıcıdan okunacaktır. Birinci karakterden başlayarak bu dizinin her bir elemanı *push* fonksiyonu ile bir yığına eklenecektir. En son yığına eklenen karakter, dizinin son elemanıdır ve yığının en üstünde yer alan ‘top’ eleman olacaktır. Yığındaki karakterler birer birer *pop* fonksiyonu ile alınıp ekrana yazdırılır. Ekrandaki çıktı dizinin elemanlarının ters sırada yazdırılması ile oluşur. Yığına en son eklenen

eleman, dizinin son elemanı, yığından ilk alınan eleman ve ekrana yazdırılan ilk karakter olacaktır. Aynı şekilde yığına sondan bir önce eklenen eleman yığından alınan ve ekrana yazdırılan ikinci karakterdir. Bu şekilde devam edildiğinde dizinin ilk elemanı ekrana en son yazdırılan eleman olacaktır ve bu istenen durumdur. Programda SIZE sembolik bir sabittir ve değeri 20 olarak belirlenmiştir. ‘top’ değişkeni global olarak tanımlanmıştır. ‘main’ fonksiyon ile *push* ve *pop* fonksiyonlarında ‘top’ değişkenine doğrudan erişilebilir.

```

1  #include <iostream>
2  using namespace std;
3  #define SIZE 20
4
5  int top=-1;
6
7  void push (char stack[], char value)
8  {
9      if(top+1<SIZE-1)
10     {
11         top++;
12         stack[top]=value;
13     }
14     else
15         cout<<"Yigin dolu. Yeni eleman eklenemez\n";
16 }

```

```

17 char pop(char stack[])
18 {
19     char temp=' ';
20     if(top!=-1)
21     {
22         temp=stack[top];
23         top--;
24         return temp;
25     }
26     else
27     {
28         cout<<"Yigin bos. Eleman silinemez\n";
29         return temp;
30     }
31 }

```

Şekil 4.18 Bir karakter dizisini tersten yazdırmak için yığın kullanımı gerçekleştiren program

```
32 int main ()
33 {
34     char a[SIZE];
35     char st[SIZE];
36     int i, count=0;
37
38     gets(a);
39     for(i=0; a[i]!='\0';i++)
40     {
41         push(st, a[i]);
42         count++;
43     }
44     for(i=1; i<=count; i++)
45         cout<<pop(st);
46     cout<<endl;
47
48     return 0;
49 }
```



ARAŞTIRALIM

postfix ve **prefix** gösterimi ile verilen matematiksel ifadelerin değerlendirilmesinde yığın kullanımını araştırınız.

BÖLÜM ÖZETİ

Bu bölümde kuyruk ve yığın veri yapıları anlatılmıştır. Önce kuyruk yapısının temel özellikleri verilmiş ve bu yapının diziler ve bağlı listeler ile uygulanması gösterilmiştir. Kuyruk ilk giren ilk çıkar temel kuralını temsil eden bir veri yapısıdır. Kuyrukta listenin bir ucundan ekleme diğer ucundan silme işlemi gerçekleşir. Sonra yığın yapısının temel kuralları anlatılmış, kuyrukta olduğu gibi diziler ve bağlı listeler kullanılarak uygulanması verilmiştir. Yığın son giren ilk çıkar prensibine göre çalışan bir veri yapısıdır. Listenin aynı ucu ekleme ve silme işlemi için kullanılır. Bu bölümde son olarak bir kullanıcından okunan karakter dizisinin tersten yazdırılma işlemi gösterilmiştir. Bu işlem yığın kullanılarak gerçekleştirilmiştir. Yığın uygulaması diziler ile yapılmış, uygulama gereği yığının her bir elemanı için karakter kullanılmıştır.

GÖZDEN GEÇİRELİM

1. Listenin bir ucundan ekleme diğer ucundan silme işlemi gerçekleşen doğrusal veri yapısı aşağıdakilerden hangisidir?
 - a) Kuyruk
 - b) Dizi
 - c) Bağlı liste
 - d) Yığın
 - e) Ağaç
2. Kuyruk ile ilgili aşağıdaki ifadelerden hangisi yanlıştır?
 - a) Dizi kullanılarak uygulanabilirler.
 - b) Bağlı liste kullanarak uygulanabilirler.
 - c) Ekleme ve silme işlemi farklı noktalarda gerçekleşir.
 - d) Dizi kullanıldığında kuyruk veri yapısının dolu olma ihtimali vardır.
 - e) Ekleme ve silme işlemi listenin aynı ucundan gerçekleştirilir.
3. Boş bir yığına sırasıyla 14, 35 ve 12 değerleri, push işlemi ile eklenmiştir. Daha sonra sırasıyla, 2 kez pop işlemi uygulanmıştır. Son olarak, 20 ile 15 değerleri push fonksiyonu ile eklenmiştir. Bu durumda yığında hangi elemanlar vardır?
 - a) 14 20 15
 - b) 12 20 15
 - c) 35 20 15
 - d) 14 12 20
 - e) 14 35 20
4. Boş bir kuyruğa sırasıyla 14, 35 ve 12 değerleri ile enqueue işlemi ile eklenmiştir. Daha sonra sırasıyla 2 kez dequeue işlemi uygulanmıştır. Son olarak, 15 ile 20 değerleri enqueue fonksiyonu ile eklenmiştir. Bu durumda kuyrukta hangi elemanlar vardır?
 - a) 14 15 20
 - b) 12 15 20
 - c) 20 15 35
 - d) 20 14 12
 - e) 35 20 14
5. Listenin aynı ucundan ekleme ve silme işlemi gerçekleşen doğrusal veri yapısı aşağıdakilerden hangisidir?
 - a) Kuyruk
 - b) Dizi
 - c) Bağlı liste
 - d) Yığın
 - e) Ağaç
6.
 - Bir programda sırasıyla birbirlerini çağıran fonksiyonlar
 - Bir markette kasanın önünde oluşan sıra
 Yukarıda verilen işlemler sırasıyla hangi veri yapıları ile ifade edilebilirler?
 - a) Kuyruk - Yığın
 - b) Ağaç - Yığın
 - c) Ağaç - Kuyruk
 - d) Yığın - Ağaç
 - e) Yığın - Kuyruk

7. Aşağıdakilerin hangisi bir kuyruğu bağlı liste ile ifade etmenin avantajlarından birisidir?

- a) Boyutunun önceden belirlenme zorunluluğu olmaması
- b) Bellekte ardışık alanlarda yerleşim
- c) Sabit belirlenmiş boyutunun olması
- d) Kolay sıralama işlemine sahip olması
- e) Kolay arama işlemine sahip olması

8. Aşağıdakilerin hangisi kuyruk veri yapısının özelliklerinden birisi değildir?

- a) Sadece bir uçtan ekleme yapılabilir
- b) Silinen eleman kuyruğun sonuna tekrar eklenir
- c) Diziler veya bağlı listeler ile uygulanabilir
- d) Aynı uç silme ve ekleme için kullanılamaz
- e) Sadece bir uçtan silme yapılabilir

9.

```
struct node{
    int v;
    struct node *p;
}*top;
```

Yığın veri yapısını uygulamak için yukarıda verilen düğüm yapısı kullanılsın. Aşağıdakilerden hangisi pop fonksiyonu içinde yazılabilecek ifadelerden birisidir?

- a) struct node *top=new struct node;
- b) top->value++;
- c) top=top->p;
- d) (top->p)++;
- e) top=top+1;

10.

```
struct node{
    int v;
    struct node *p;
}*top;
```

Yığın veri yapısını uygulamak için yukarıda verilen düğüm yapısı kullanılsın. Aşağıdakilerden hangisi push fonksiyonu içinde yazılabilecek ifadelerden birisidir?

- a) struct node *newNode=new struct node;
- b) top->value++;
- c) top=top->p;
- d) (top->p)++;
- e) top=top+1;

Yanıt Anahtarı: 1-A 2-E 3-A 4-B 5-D 6-E 7-A 8-B 9-C 10-A

Kaynakça

Deitel, P., Deitel, H. (2017). C++ How to Program. 10th Edition. Pearson.

Thareja R. (2014) Data Structures Using C, 2nd Edition, Oxford University Press.

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 5

AĞAÇLAR

ANAHTAR KAVRAMLAR

Ağaç, İkili Ağaç, İkili Arama Ağacı, AVL Ağacı



ÖĞRENME HEDEFLERİ

-  1 Ağaç veri yapısının temellerini bilir ve uygulayabilir
-  2 İkili ağaç yapısını farklı veri yapıları oluşturmak için kullanabilir
-  3 İkili arama ağaçlarını tanımlayabilir ve uygulayabilir
-  4 AVL ağaçlarını tanımlar ve uygulayabilir

GİRİŞ

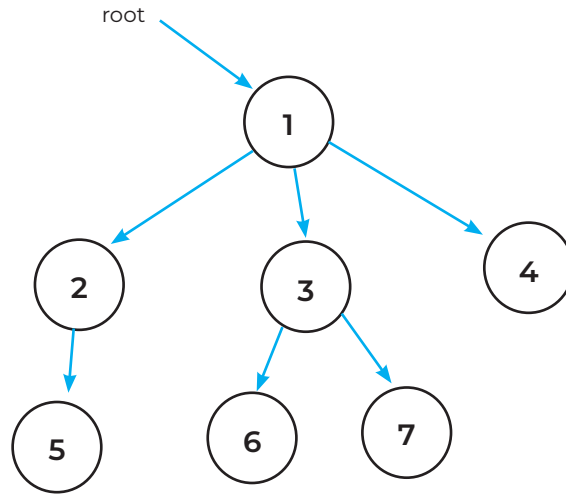
Bu bölümde ağaç veri yapısı anlatılacaktır. Ağaç veri yapısı doğrusal olmayan bir veri yapısıdır ve düğümler arasında hiyerarşik bir ilişki vardır. Bu veri yapısında kök (root) düğüm dışında tüm düğümlerin bir ebeveyn (parent) düğümü vardır. Bir ebeveyn düğüme bir dal (edge) ile bağlı olan düğüme çocuk (child) denir. Çocuğu olmayan düğümlere yaprak (leaf) düğüm adı verilir. Eğer bir ağaç veri yapısında bir düğümün en fazla iki çocuğu olabiliyorsa bu ağaç yapısına ikili (binary) ağaçlar denir. İkili ağaçların belirli kurallar çerçevesinde tanımlanan farklı biçimleri vardır. İkili arama ağaçları ve AVL ağaçları farklı formlarda tanımlanmış ikili ağaçlardır. Bu bölümde detaylı olarak bu ağaç tipleri üzerinde gerçekleştirilecek işlemler anlatılacaktır.



1. AĞAÇLAR

Ağaç veri yapısı düğümler arasında ebeveyn-çocuk ilişkisi kuran hiyerarşik bir veri yapısıdır. Bir ağaç veri yapısı kök düğümü ile başlar. Kök düğümün ebeveyn düğümü bulunmamaktadır. Kök düğüm dışında tüm düğümlerin bir ebeveyn düğümü vardır. Bir düğümün birden fazla ebeveyni olamaz. Bu özelliği dolayısıyla ağaç veri yapısı çizgilerden ayrılır. Çizgelerde her bir düğüm birden fazla düğüm ile bağlantı oluşturabilir. Bu nedenle ağaç veri yapısı çizgelerin bir alt kümesi, bir özel durumu olarak değerlendirilebilir. Kök düğümünden başlayarak ağaç veri yapısında yer alan tüm düğümlere ulaşmak mümkündür. Genel ağaç yapısında her bir düğümün bir veya birden fazla çocuğu olabilir. Çocuğu olmayan düğümlere yaprak düğüm adı verilir. Ağaç içerisinde yapraklar son düğümler olarak değerlendirilir.

Şekil 5.1'de verilen ağaç yapısında 7 düğüm vardır. Kök düğümün 3 çocuğu vardır. Burada verilen her bir çocuk düğüm (2, 3 ve 4) aynı zamanda kendi alt ağaçlarının kök düğümüdürler. Dolayısı ile bir ağaç veri yapısı, özyinelemeli olarak her bir düğümün bir alt ağacın kök düğümü olduğu yaklaşımı ile de tanımlanabilir. Ağaç veri yapısının yüksekliği ağacın kökünden en altta yer alan yaprak düğüme kadar ulaşan yolun uzunluğu olarak tanımlanır. Bir düğümün derinliği ise kök düğüme olan uzaklığıdır. Kök düğümün derinliği 1 olarak kabul edilir. Şekil 5.1'de yer alan ağacın yüksekliği 2'dir. Aynı ağaçta yer alan 6 düğümünün derinliği 3, 4 düğümünün derinliği 2'dir. Aynı ebeveyne sahip düğümler kardeş (sibling) düğümler olarak değerlendirilirler.



Şekil 5.1. Örnek bir ağaç veri yapısı

Bir ağaç yapısında eğer bir ebeveyn en fazla iki çocuk ile sınırlanmışsa bu tür ağaç veri yapısına ikili ağaç yapısı ismi verilir. Bu tür ağaç yapısında

düğümlerde hiç çocuk olmayabilir, bir çocuk olabilir veya iki çocuk olabilir. İkili ağaç yapısında kullanılan düğüm yapısı Şekil 5.2'de verilmiştir.



ARAŞTIRALIM

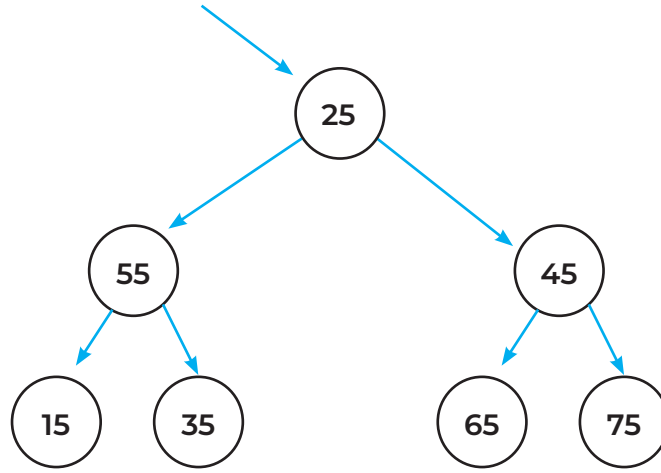
Bu bölümde ağaçlar için bağlı listeler kullanılmıştır. Ağaçların uygulanmasında dizilerin kullanımını araştırılmalıdır.

Şekil 5.3'te 7 düğümden oluşan örnek ikili ağaç yapısı verilmektedir. Bu ağaç yapısında düğümlerde yer alan değerler arasında herhangi bir düzen verilmemiştir. Ancak sonraki alt bölümler-

de düğümlerde yer alan veriler arasında belirli düzenlerin olduğu ikili ağaç tipleri de gösterilecektir.

```
1 struct treeNode
2 {
3     int value;
4     struct treeNode *leftChild;
5     struct treeNode *rightChild;
6 };
```

Şekil 5.2. İkili ağaç yapısında kullanılan düğüm yapısı



Şekil 5.3. Örnek bir ikili ağaç veri yapısı



DİKKAT EDELİM

Yaprak düğümlerde sol çocuk ve sağ çocuk düğümleri göstermek için kullanılan işaretçiler NULL değerine sahiptir.

Bu düğüm yapısında veri olarak sadece tam sayı kullanılmıştır. Ayrıca sol çocuğu gösteren leftChild işaretçisi ve sağ çocuğu gösteren

rightChild işaretçisi bu düğümün diğer elemanlarıdır. Yaprak düğümlerde her iki işaretçinin değeri NULL olarak yer alır.



ARAŞTIRALIM

Genel bir ağaç yapısını, ikili ağaç yapısına dönüştürme yöntemini araştıralım.

Bir ikili ağaç veri yapısında en önemli işlemlerden birisi ağaç üzerinde gezinme işlemidir. Temel olarak üç farklı gezinme yöntemi vardır.

1. Preorder Gezinme
2. Inorder Gezinme
3. Postorder Gezinme

```

1 void preorder(struct treeNode *tree)
2 {
3     if(tree!=NULL)
4     {
5         cout<<tree->value<<' ';
6         preorder(tree->leftChild);
7         preorder(tree->rightChild);
8     }
9 }

```

Şekil 5.4. Preorder gezinme fonksiyonu

Preorder gezinme yönteminde kök düğüm çocuklardan önce ziyaret edilir veya bir başka ifade ile işlem görür (Şekil 5.4). Daha sonra sol çocuk ve son olarak sağ çocuk ziyaret edilir. Sol veya sağ çocuk ziyaret edildiğinde bu düğümler, bir alt ağacın kök düğümleri olarak düşünülürler. Dolayısı ile özyinelemeli olarak çağrılan **preorder** gezinme fonksiyonu bu düğümleri kök olarak kabul eder. Şekil 5.3'te verilen ikili ağaçta yer alan düğümlerin **preorder** gezinme ile ziyaret sırası aşağıdaki gibidir.

25 55 15 35 45 65 75

Öncelikli olarak **preorder** gezinme fonksiyonuna kök 25 düğümü gönderildiğinde bu değer ekrana yazdırılır. Sonra sol çocuk ziyaret edilir. Bu düğüm sol alt ağacın kök düğümüdür. Bu nedenle ekrana yazdırılır ve bu düğümün sol çocuğu olan 15 düğümüne gelinir. 15 ekrana yazdırılır. 15 değerine sahip olan düğümün önce sol çocuğuna sonra sağ çocuğuna bakılır. Her ikisi de NULL olduğu için, 15 değerinin kök olduğu ikili ağacın işlemi tamamlanır. Bu durumda işlem sırası, kök değeri 35 olan alt ağaca geçer ve bu süreç bu şekilde devam eder.

```

1 void inorder (struct treeNode *tree)
2 {
3     if(tree!=NULL)
4     {
5         inorder(tree->leftChild);
6         cout<<tree->value<<' ';
7         inorder(tree->rightChild);
8     }
9 }

```

Şekil 5.5. Inorder gezinme fonksiyonu

Inorder gezinme işleminde kök düğüm ziyaret edilmeden önce sol çocuk ziyaret edilir (Şekil 5.5). Sol alt ağaçta yer alan işlemler tamamlandıktan sonra kök düğüm, daha sonra sağ alt düğüm işlem görür. Tabii ki sol çocuk veya sağ çocuk ziyaret edildiğinde bu düğümler ilgili alt ağacın

kök düğümü olarak işlem görürler. Özyinelemeli olarak çağrılan **inorder** fonksiyonu bu düğümleri kök olarak alır. Şekil 5.3'te verilen ikili ağaçta yer alan düğümlerin **inorder** gezinme ile ziyaret sırası aşağıdaki gibidir.

15 55 35 25 65 45 75

```

1 void postorder (struct treeNode *tree)
2 {
3     if(tree!=NULL)
4     {
5         postorder(tree->leftChild);
6         postorder(tree->rightChild);
7         cout<<tree->value<<' ';
8     }
9 }

```

Şekil 5.6. Postorder gezinme fonksiyonu

Postorder gezinme işleminde önce sol alt ağaç, sonra sağ alt ağaç ve son olarak kök düğüm ziyaret edilir (Şekil 5.6). Şekil 5.3'te yer alan ikili ağaç üzerinde **postorder** gezinme gerçekleştirdiğimizde elde edilen gezinme sırası aşağıdaki gibidir.

15 35 55 65 75 45 25

Eğer **inorder** gezinme sonucu ile beraber **pre-order** veya **postorder** gezinme sonucu verilmiş ise ikili ağacı tekrar oluşturmak mümkündür.



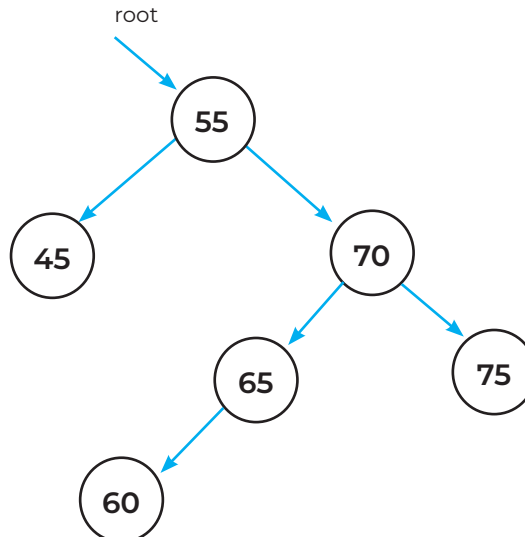
ARAŞTIRALIM

Preorder veya postorder gezinme ile inorder gezinme sonuçlarının beraber verildiği durumda ikili ağaç yapısını oluşturmayı araştıralım.

2. İKİLİ ARAMA AĞAÇLARI (BINARY SEARCH TREES)

Genel ikili ağaç yapısında, düğümlerde yer alan değerlerin büyüklükleri arasında herhangi bir ilişki yoktur. İkili arama ağaçları ikili ağaç yapısının özel bir formudur ve düğümlerin değerleri arasında büyüklük-küçüklük ilişkisi söz konusudur. Sıralı bir listedeki arama işlemini verimli bir şekilde gerçekleştirmek için geliştirilmiş bir yapıdır. Arama, ekleme ve silme işlemleri temel olarak **O(logn)** sürede gerçekleştirilir. Bir düğümün değeri, sol alt ağaçtaki tüm değerlerden büyük,

sağ alt ağaçtaki tüm değerlerden küçük veya eşit olmalıdır. Şekil 5.7'de örnek bir ikili arama ağacı verilmiştir. Kök düğümde yer alan 55 değeri sol alt ağaçtaki her değerden büyük, sağ alt ağaçtaki her değerden küçüktür. Aynı şekilde bu kural ikili arama ağacındaki tüm düğümler için geçerlidir. Sağ alta ağaçta 70 değeri kendisinin sol alt ağacındaki tüm değerlerden büyük, sağ alt ağacındaki tüm değerlerden küçüktür.



Şekil 5.7. Örnek bir ikili arama ağacı

Şimdi de ikili arama ağaçlarında arama işleminin nasıl gerçekleştirildiğine bakalım. Arama işlemi için önce aranan değer kökte yer alan değer ile karşılaştırılır. Eğer aranan değer kök değerinden küçükse sağ alt ağaç elenir ve arama işlemi sol alt ağaç ile devam eder. Aksi takdirde aranan değer kök değerinden büyükse sol alt ağaç elenir arama işlemi sağ alt ağaç ile devam eder. Eğer kök değeri aranan değer ile eşit ise arama işlemi başarıyla bir şekilde sonlandırılır ve ilgili düğümün adresi çevrilir. Bu süreç özyinelemeli olarak bu şekilde devam eder. Eğer alt ağaçlar ile devam etme sürecinde NULL değerine ulaşırsa aranan

değerin ikili arama ağacında olmadığı anlaşılır. Örnek olarak 73 değerini Şekil 5.7'de arayalım. 73 değeri kökte yer alan 55 değeri ile karşılaştırılır. 73 55'ten büyük olduğu için arama işlemine 55 düğümünün sağ alt ağacı ile devam edilir. 73 değeri 70 ile karşılaştırılır ve büyük olduğu için 70 düğümünün sağ alt ağacı ile devam edilir. İlgili sağ alt ağacın kök değeri 75 ile karşılaştırılır ve bu değerden küçük olduğu için sol alt ağaca devam edilir. Ancak 75'in sol alt ağacı NULL'dır ve 73 değerinin ikili arama ağacında olmadığı anlaşılır. Arama işlemi için yazılan **search** fonksiyonu Şekil 5.8'de verilmiştir.

```

1 struct treeNode* search(struct treeNode *tree, int sValue)
2 {
3     if(tree==NULL || tree->value == sValue)
4         return tree;
5     else
6     {
7         if(sValue < tree->value)
8             return search(tree->leftChild, sValue);
9         else
10            return search(tree->rightChild, sValue);
11     }
12 }
```

Şekil 5.8. İkili arama ağacında bir değeri arama fonksiyonu

İkili arama ağacına yeni bir değer eklemek için arama işleminde olduğu gibi kök düğümdeki değer ile eklenmek istenen değer karşılaştırılır. Eğer eklenmek istenen değer küçükse sol alt ağaç ile büyükse sağ alt ağaç ile devam edilir. Bu süreç

NULL bir sol işaretçi veya sağ işaretçiye ulaşıncaya kadar devam eder. Şekil 5.9'da ikili arama ağacına bir değer eklemek için yazılan **insert** fonksiyonu verilmiştir.

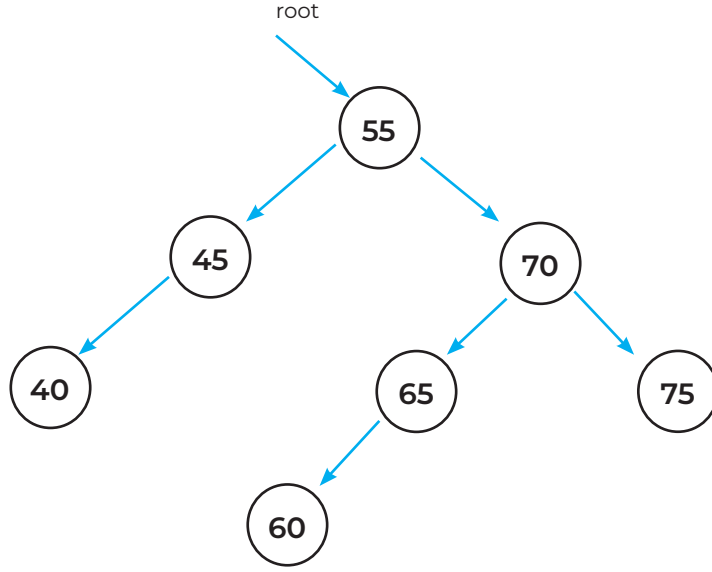
```

1 void insert(struct treeNode *tree, int nValue)
2 {
3     if(tree==NULL)
4     {
5         tree=new treeNode;
6         tree->value=nValue;
7         tree->leftChild=NULL;
8         tree->rightChild=NULL;
9     }
10    else
11    {
12        if(nValue < tree->value)
13            insert(tree->leftChild, nValue);
14        else
15            insert(tree->rightChild, nValue);
16    }
17 }
```

Şekil 5.9. İkili arama ağacına bir değeri ekleme fonksiyonu

Şekil 5.7'de yer alan ikili arama ağacına 40 değerini ekleyelim. 40 kök düğümde yer alan 55 değerinden küçük olduğu için işleme sol alt ağaç ile devam edilir. 40 sol alt ağacın kök düğümünde yer alan 45'ten de küçüktür. Dolayısıyla işleme 45 düğümünün sol alt ağacı ile devam edilir. Bu

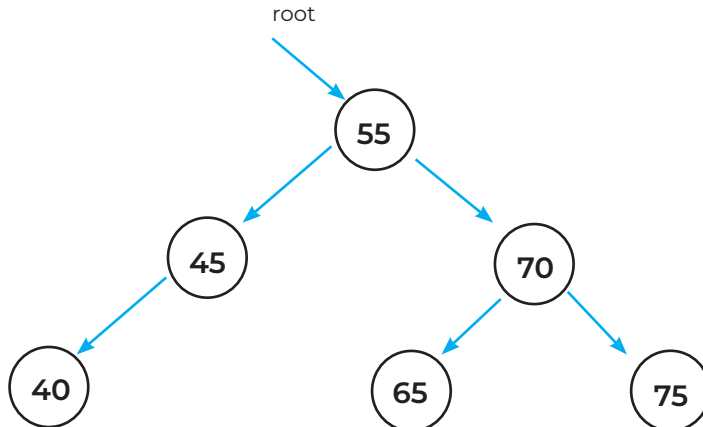
düğümün sol alt ağacı gösteren sol işaretçi değeri NULL olduğu için yeni düğümü ekleme noktası bulunmuş olur. 45 düğümünün sol çocuğu olarak 40 düğümü ikili arama ağacına eklenir. Yeni ikili arama ağacı Şekil 5.10'da verilmiştir.



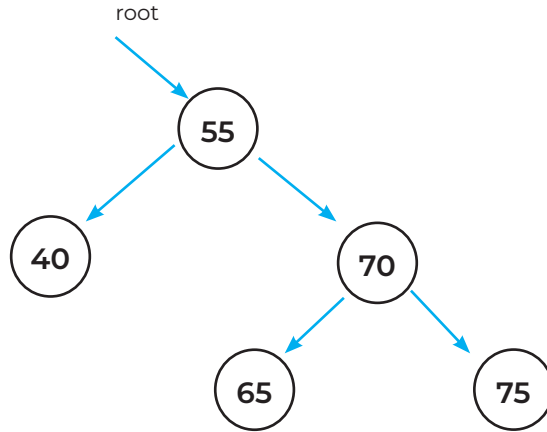
Şekil 5.10. Şekil 5.7’de yer alan ikili arama ağacına 40 düğümünün eklenmesi ile oluşan yapı

Bir düğümü ikili arama ağacından silmek için önce ilgili düğümün ağaçta arama işlemi gerçekleştirilir. Eğer düğüm ağaç yapısı içinde mevcutsa silme işlemi gerçekleştirilir. Bir ikili arama ağacından bir düğümü silme işlemi için üç farklı durum söz konusudur. Bu durumlar silinecek düğümün kaç tane çocuğunun olması ile ilgilidir. Bir düğümün hiç çocuğu olmayabilir, bir çocuğu olabilir veya iki çocuğu olabilir. En kolay silme işlemi düğümün hiç çocuğunun olmaması durumudur. Düğüm ağaç yapısından uzaklaştırılır. Örneğin Şekil 5.10’da yer alan ikili arama ağacından 60 değerine sahip düğümü silmeye çalışalım. Bu düğümün çocuğu yoktur ve düğüm doğrudan farklı bir işleme gerek kalmadan silinir. Elde edilen ikili arama ağacı Şekil 5.11’de verilmiştir. İkinci durum ise sadece bir çocuğu olan düğümün silinme durumudur. Bu durumda silinecek

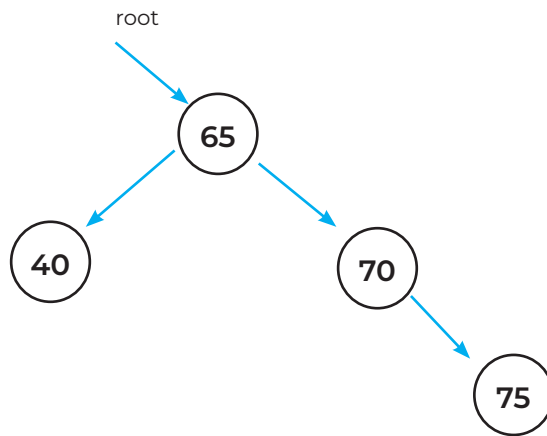
düğümün yerine tek çocuğu yerleştirilir. Şekil 5.11’deki ikili arama ağacından tek çocuğu olan 45 düğümü silindiğinde elde edilen ağaç yapısı Şekil 5.12’de verilmektedir. Son durum iki çocuğa sahip bir düğümün ikili arama ağacından silinmesidir. Bu durumda silinecek düğümün halefi (successor) bulunur ve silinecek düğümün yerine yazılır. Şekil 5.12’de verilen ikili arama ağacından kök düğümün silinme işlemi bu duruma örnektir. 55 değerine sahip olan düğümün iki çocuğu vardır. Bu düğümün halefi sağ alt ağaçta yer alan en küçük değere sahip düğümdür. Bir başka ifade ile sağ alt ağaçta en solda yer alan düğümdür. Bu düğüm ilgili örnekte 65 değerine sahiptir. 55 değerine sahip düğüm silinir ve 65 değerine sahip halefi ikili arama ağacında onun yerini alır. Bu durum Şekil 5.13’te yer almaktadır.



Şekil 5.11. İkili arama ağacından çocuğu olmayan 60 düğümünün silinmesi ile oluşan yapı



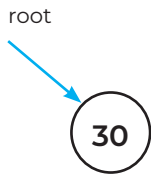
Şekil 5.12. İkili arama ağacından bir çocuğu olan 45 düğümünün silinmesi ile oluşan yapı



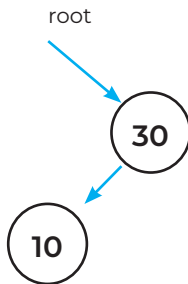
Şekil 5.13. İkili arama ağacından iki çocuğu olan 55 düğümünün silinmesi ile oluşan yapı

Bir listedeki değerlerin sırasıyla ikili arama ağacına yerleştirilmesi işleminde değerlerin listedeki yerine göre farklı ağaçlar oluşabilir. Örnek olarak 30, 10, 20 ve 40 değerlerinin bu sırada ikili arama ağacına yerleştirildiğini düşünelim. Ağaç boşken ilk düğüm 30 kök olarak eklenir (Şekil 5.14. (a)). Listedeki ikinci değer 10, kök düğümde yer alan 30 değerinden küçüktür ve sol çocuk olarak eklenir (Şekil 5.14. (b)). Bir sonraki değer

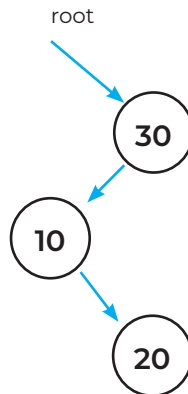
20, kök düğümün değeri 30 dan küçüktür. Yeni değerinin ekleneceği nokta için sol alt ağaç ile devam edilir. Sol alt ağacın kök değeri 10 ile yeni değer 20 karşılaştırılır. 20 bu değerden büyüktür ve 10 değerinin sağ çocuğu olarak ağaca eklenir (Şekil 5.14. (c)). Son değer 40 kök 30 değerinden daha büyüktür ve kökün sağ çocuğu olarak eklenir (Şekil 5.14. (d)).



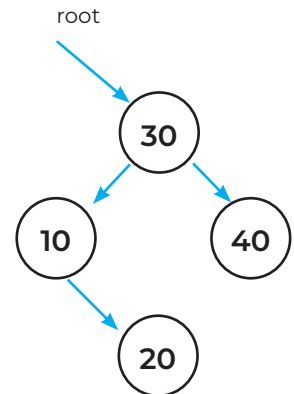
(a)



(b)



(c)



(d)

Şekil 5.14. İkili arama ağacına 4 değer sıralı olarak eklenmesi

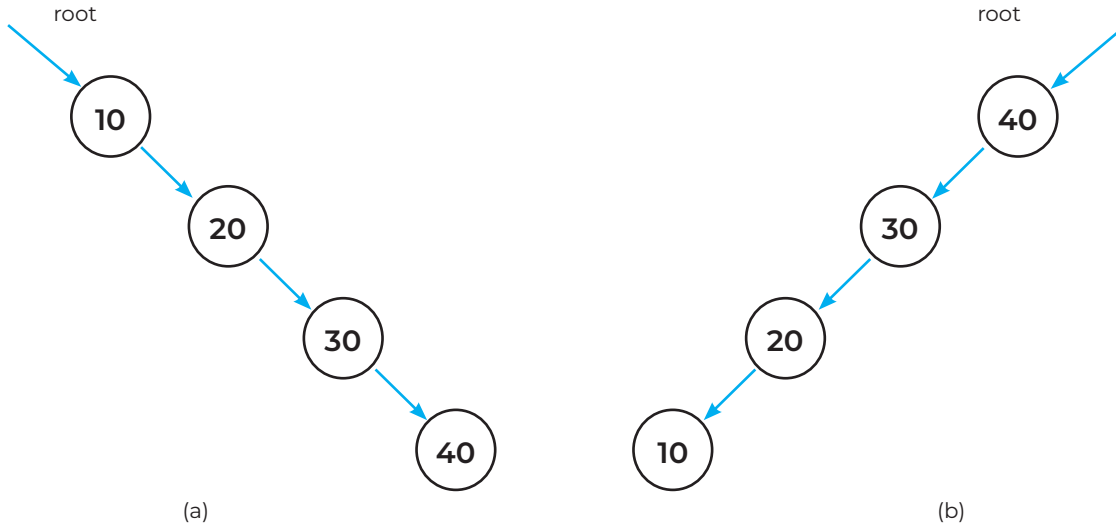
Aynı sayılar eğer farklı sırada, ikili arama ağacına yerleştirilirse Şekil 5.14'te yer alandan farklı bir ağaç yapısı elde edilir. Eğer listede yer alan elemanlar 10, 20, 30 ve 40 sıralaması ile ikili arama ağacına eklenirse Şekil 5.15. (a)'da yer alan ağaç elde edilir. Eğer listedeki elemanların düzeni 40, 30, 20 ve 10 şeklinde azalan sırada verilmişse Şekil 5.15. (b)'de yer alan ikili arama ağacı oluşur. Buradaki örneklerden görüldüğü gibi ikili arama ağaçları doğrudan elemanların eklenme sırasına bağlıdır. Eğer elemanlar arasında artan veya azalan bir sıralama söz konusu ise Şekil 5.15'teki gibi

sırasıyla sağa dayalı (right-skewed) ve sola dayalı (left-skewed) ikili arama ağaçları ortaya çıkabilir. Sağa dayalı ve sola dayalı ikili arama ağaçları Şekil 5.15'te de görüleceği gibi doğrusal yapılar oluşturur. Bu doğrusal yapılar, ekleme ve arama işlemleri için ikili arama ağaçlarının, hedeflenen çalışma zamanına ulaşmasını ($O(\log n)$) engeller. Her iki yapıda doğrusal çalışma zamanına sahiptir. Bazı durumlarda ikili arama ağaçlarının beklenen çalışma zamanı performansını göstermemesi AVL ağaçlarının oluşturulmasına yol açmıştır.



DİKKAT EDELİM

İkili arama ağaçlarında elemanların ağaca eklenme sırasına göre sola dayalı veya sağa dayalı doğrusal yapılar ortaya çıkabilir.



Şekil 5.15. İkili arama ağacına 4 değer farklı sıralarda eklenmesi

3. AVL AĞAÇLARI

Bir önceki kısımda verilen örnekte bazı durumlarda özellikle sıralı listelerde ikili arama ağaçlarının doğrusal yapılar oluşturabileceği görülmüştü. Sağa dayalı ve sola dayalı ikili arama ağaçları olarak isimlendirilen bu durumlar ikili arama ağaçlarının verimli şekilde çalışmasını engellemektedir. Bu duruma çözüm olması amacıyla

AVL ağaçları geliştirilmiştir. AVL ağaçlarındaki temel amaç sol ve sağ alt ağaçlar arasındaki yükseklik farkının belirli bir sınır çerçevesinde tutulmasıdır. İki alt ağaç arasındaki yükseklik farkı en fazla 1 olabilir. Bu değere denge faktörü (balance factor) adı verilir. Alt ağaçlarının yüksekliği eşit ise denge faktörü 0, sol alt ağaç yüksekliği 1

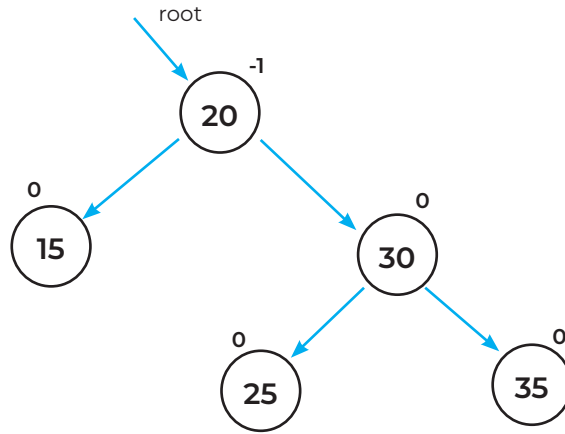
fazla ise denge faktörü 1 ve 1 eksi denge faktörü -1 olarak hesaplanır. -1, 0 ve 1 denge faktörü değerleri AVL ağacındaki düğümler için kabul edilebilir değerlerdir. Bu değerlerden farklı bir denge faktörüne sahip bir düğüme sahip ağaç AVL ağacı olarak değerlendirilmez. Şekil 5.16'da 5 düğümden oluşan bir AVL ağacı yer almaktadır. Düğümlerin yanında denge faktörü değerleri

de verilmektedir. Burada önemli olan nokta AVL ağaçlarının aynı zamanda ikili arama ağacı özelliğine de sahip olmalarıdır. Bu özelliğe ek olarak denge faktörü kavramı ile yükseklik dengeli (height-balanced) bir ağaç oluştururlar. Bu şekilde ikili arama ağaçlarının doğrusal yapı oluşturma ihtimali ortadan kaldırılmış olur.



ARAŞTIRALIM

AVL ağaçları dışında ikili arama ağaçlarının daha verimli kullanılmasını sağlayan farklı ağaç tiplerini araştırınız.



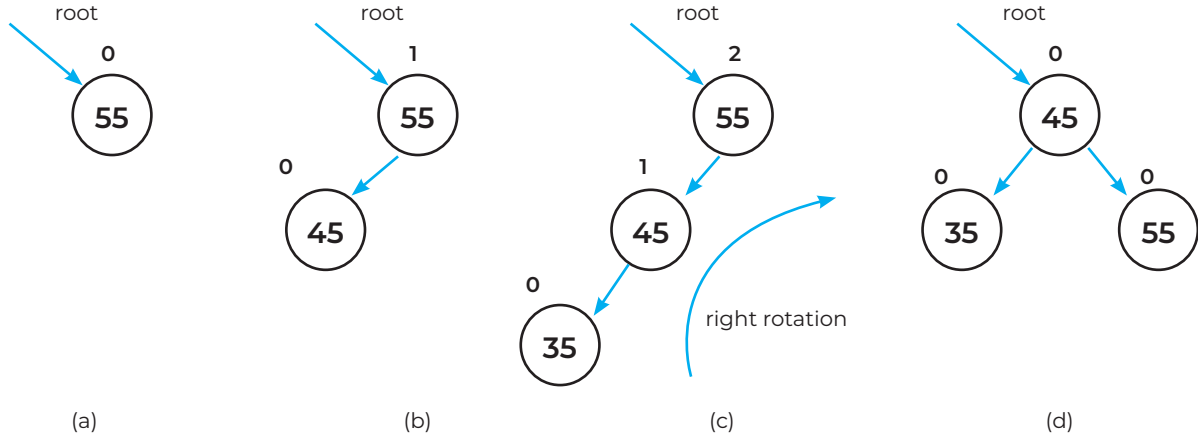
Şekil 5.16. 5 düğümden oluşan örnek bir AVL ağacı

AVL ağacında arama ve ekleme işlemleri, ikili arama ağacındaki işlemlere benzer şekilde gerçekleştirilir. Arama işlemi birebir aynıdır. Ancak ekleme işleminden sonra düğümlerin denge faktörlerinde bozulma olabilir. Bir başka ifade ile denge faktörü -1, 0 ve 1'den farklı değerlere sahip olabilir. Bu durumda denge faktörlerinde oluşan bozulmayı düzeltmek için sola rotasyon (left rotation) veya sağa rotasyon (right rotation) işlemleri gerçekleştirilir. Şekil 5.17'de bu duruma bir örnek verilmiştir. Önce 55 değeri boş bir AVL ağacına eklenmiştir. Bu ilk düğümün denge faktörü 0'dır. Daha sonra 45 eklenmek istendiğinde 55 değerinden küçük olduğu için bu düğümün soluna ekleme işlemi gerçekleşir. Tabii ki ekleme

işleminin yapıldığı noktadan kök düğüme kadar ulaşan yoldaki tüm düğümlerin denge faktörleri güncellenir. Güncelleme sonrasında 55 değerine sahip düğümün denge faktörü 1 olarak hesaplanır. Sol alt ağacın yüksekliği sağ alt ağacın yüksekliğinden 1 fazladır. Bu durumda AVL özelliği bozulmaz. Yeni eklenen 45 düğümünün denge faktörü 0'dır. Görüleceği gibi yeni eklenen düğümler yaprak düğüm olarak eklendikleri için sol çocuk ve sağ çocuk işaretçi değerleri NULL'dır ve iki alt ağaç arasındaki yükseklik farkı 0'dır. Şekil 5.17 (b)'deki AVL ağacına 35 eklenmek istendiğinde, 35 değeri kök düğümün değeri 55'ten ve onun sol alt ağacının kök düğümünün değeri 45'ten küçüktür. Bu nedenle 45 düğümünün sol çocu-

ğu olarak eklenir. Denge faktörleri güncellenirler ve kök düğümün denge faktörünün 2 olduğu görülür. Sol alt ağacın yüksekliği sağ alt ağacın yüksekliğinden 2 fazladır. Bu durum AVL özelliğini bozar. Çözüm için sağ rotasyon kullanılır. 45

değerine sahip düğüm kök düğüm olur. Önceki kök düğüm yeni kök düğümün sağ çocuğu olarak sağa hareket ettirilir. Bu durum Şekil 5.17'de gösterilmektedir.



Şekil 5.17. 55, 45 ve 35 değerleri ile bir AVL ağacı oluşturulması



ARAŞTIRALIM

AVL ağaçlarından bir elemanın silinmesi işleminde oluşabilecek denge faktörü değişikliklerini ve çözüm yollarını araştırınız.

BÖLÜM ÖZETİ

Bu bölümde öncelikli olarak genel ağaç veri yapısı anlatılmıştır. Ağaçlar elemanlar arasında hiyerarşik bir ilişkiye sahip doğrusal olmayan veri yapılarıdır. Kök düğüm dışında ağaçta yer alan her bir düğümün bir ebeveyn düğümü vardır. Bir düğümün birden fazla ebeveyn düğüme sahip olması mümkün değildir. Bu yönü dolayısıyla ağaçlar, çizgilerden ayrılırlar. Bir düğümün en fazla iki çocuk ile sınırlandığı ağaçlara ikili ağaçlar denir. İkili ağaçlarda tüm düğümleri ziyaret etmek için üç farklı gezinme yöntemi vardır. Preorder gezinmede kök düğüm sol ve sağ alt ağaçtan önce, postorder gezinmede ise alt ağaçlardan sonra ziyaret edilir. Bir diğer yöntem inorder gezinmede önce sol alt ağaç ziyaret edilir daha sonra kök düğüm ve son olarak sağ alt ağaç ziyaret edilir. İkili arama ağacı, ağacı oluşturan düğümlerin değerleri arasında büyüklük-küçüklük ilişkisinin olduğu bir yapıdır. Kökte yer alan düğümün değeri tüm sol alt ağaçta yer alan değerlerden büyük, sağ alt ağaçta yer alan düğümlerin değerlerinden küçük veya eşittir. İkili arama ağaçları arama ve ekleme gibi veri yapısı işlemlerinin sıralı bir listede verimli bir şekilde gerçekleştirilmesi için oluşturulan bir yapıdır. Listedeki elemanların ikili arama ağacına eklenme sırası ağacın yapısını belirler. Bu nedenle bazı durumlarda sola dayalı ve sağa dayalı ağaç yapıları ortaya çıkabilir. Bu duruma çözüm için AVL ağaçları geliştirilmiştir. AVL ağaçları, ikili arama ağaçlarının özelliğine ek olarak denge faktörü kullanarak yükseklik dengeli bir yapı oluşturmaktadır.

GÖZDEN GEÇİRELİM

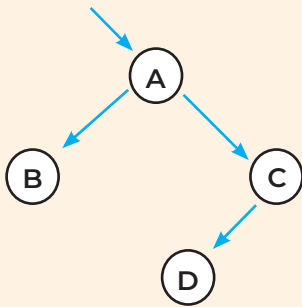
1. Aşağıdakilerden hangisi ağaç veri yapısı ile ilgili doğru bir ifadedir?

- a) Düğümlerin birden fazla ebeveyni olabilir.
- b) Yaprak düğümlerin bir çocuğu olabilir.
- c) Kök düğümün ebeveyni yoktur.
- d) Bir düğümde çocuk sayısı kadar ebeveyn vardır.
- e) Yaprak düğümlerin ebeveyni yoktur.

2. Ağaç veri yapısı ile ilgili aşağıda verilen ifadelerden hangisi yanlıştır?

- a) Ağaç veri yapısı doğrusal olmayan bir veri yapısıdır.
- b) Ağaçtaki tüm düğümlerin ebeveyni vardır.
- c) Bir düğümün iki ebeveyni olamaz.
- d) Bir düğümün birden fazla çocuğu olabilir.
- e) Yaprak düğümlerde çocukları gösteren işaretçiler NULL değerine sahiptir.

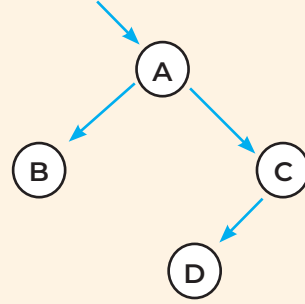
3.



Yukarıda verilen ikili ağaç yapısında aşağıdakilerden hangisi postorder gezinme ile elde edilir?

- a) B-D-C-A
- b) B-A-C-D
- c) D-C-B-A
- d) C-B-A-D
- e) D-B-C-A

4.



Yukarıda verilen ikili ağaç yapısında aşağıdakilerden hangisi inorder gezinme ile elde edilir?

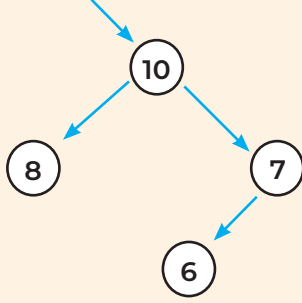
- a) D-C-B-A
- b) B-C-D-A
- c) B-C-A-D
- d) A-B-C-D
- e) B-A-D-C

5. Aşağıdaki gezinme yöntemlerinden hangisi ikili arama ağacında artan düzende sıralı bir liste üretir?

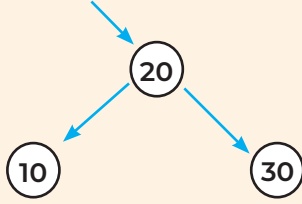
- a) levelorder gezinme
- b) postorder gezinme
- c) inorder gezinme
- d) preorder gezinme
- e) Sıralı liste oluşmaz

6. Aşağıdakilerden hangisi geçerli bir ikili arama ağacıdır?

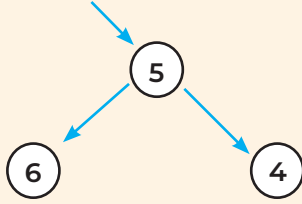
a)



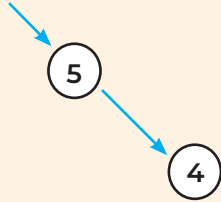
b)



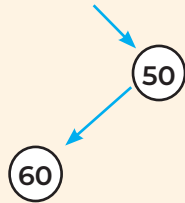
c)



d)



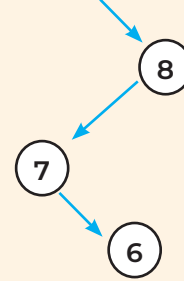
e)



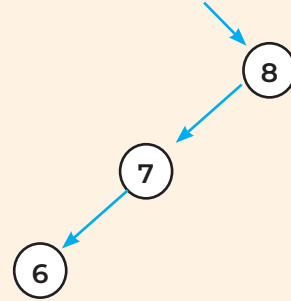
7. 8 – 7 – 6

Aşağıdaki verilen rakamlar verildiği sırada ikili bir arama ağacına yerleştirildiğinde aşağıdakilerden hangisi oluşur?

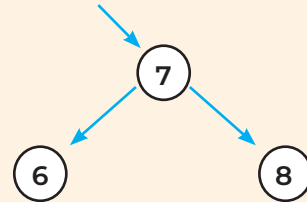
a)



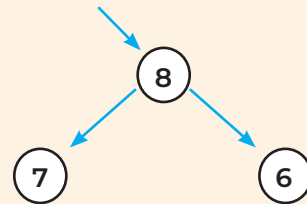
b)



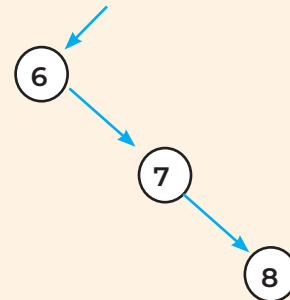
c)



d)



e)



8. Bir AVL ağacının ikili arama ağacından temel farkı aşağıdakilerden hangisidir?

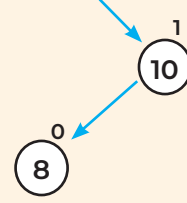
- a) Düğümlerdeki çocuk sayısı
- b) Düğümler arasındaki büyüklük ilişkisi
- c) Kırmızı düğümler
- d) Denge faktörü
- e) Siyah düğümler

9. Aşağıdakilerden hangisi AVL ağaçları için söylenemez?

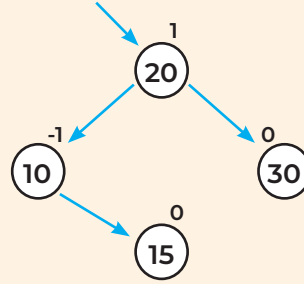
- a) Düğümlerde yer alan veriler arasında büyüklük küçüklük ilişkisi vardır
- b) 2 veya -2 denge faktörüne sahip düğümler geçerli bir AVL ağacında yer alamaz.
- c) Sağ ve sol alt ağaçlar arasındaki yükseklik farkı ikiden fazla olabilir.
- d) Bir düğümün en fazla iki çocuğu vardır.
- e) İkili arama ağaçlarının daha verimli bir formu olarak geliştirilmiştir.

10. Aşağıdakilerden hangisi geçerli bir AVL ağacı değildir?

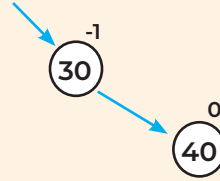
a)



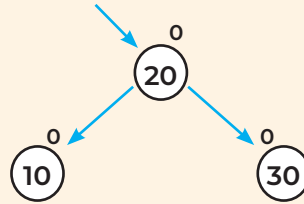
b)



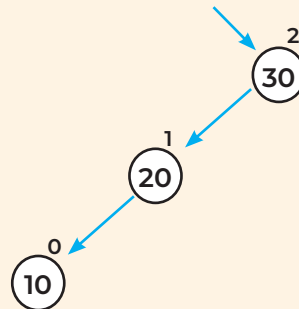
c)



d)



e)



Yanıt Anahtarı: 1-C 2-B 3-A 4-E 5-C 6-B 7-B 8-D 9-C 10-E

Kaynakça

Deitel, P., Deitel, H. (2017). C++ How to Program. 10th Edition. Pearson.

Thareja R. (2014) Data Structures Using C, 2nd Edition, Oxford University Press.

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 6

ÖZETLEME (HASH) TABLolarI

ANAHTAR KAVRAMLAR

Özetleme Tabloları, Çatışma,
Açık Adresleme, Ayırık Zincirleme



ÖĞRENME HEDEFLERİ

- 1 Özetleme tablolarının kullanılma amacını bilir.
- 2 Özetleme tablolarını farklı yöntemler kullanarak oluşturabilir.
- 3 Özetleme tablolarında oluşabilecek çatışmaların farkında olur.
- 4 Çatışma problemini çözmek için açık adresleme kullanabilir.
- 5 Ayırık zincirleme kullanarak çatışma problemini çözebilir.

GİRİŞ

Arama, bir veri yapısı üzerinde en fazla kullanılan işlemlerden birisidir. Arama işlemi ile beraber ekleme ve silme işlemlerini de bu bölüme kadar işlenen tüm veri yapılarından daha hızlı bir şekilde gerçekleştirmek için özetleme (hash) tabloları geliştirilmiştir. Bu veri yapısında hedeflenen çalışma süresi, yukarıda ifade edilen temel işlemler için $O(1)$ ile ifade edilen sabit bir zamandır. Hash tablosuna bir anahtar değeri eklemek için öncelikli olarak bu değerden bir indis üretilir. Bu indis değeri hash fonksiyonları yardımıyla hesaplanır. Hash fonksiyonları üretilecek indis değerlerini tüm tablo üzerinde dengeli bir şekilde dağıtmaya çalışır. Bu nedenle hash fonksiyonları kullanarak, bir anahtar değeri hash tablosuna ekleme işlemi kararlı bir süreçtir. Ekleme için kullanılan hash fonksiyonu arama işlemi için de kullanılacaktır ve bu nedenle fonksiyonun aynı anahtar değer için aynı indis değerini üretmesi beklenir. Hash tablosuna yeni bir değer ekleme işleminde bazen ilgili hash tablosu alanı dolu olabilir. Bu durum çatışma olarak değerlendirilir. Bu bölümde farklı hash fonksiyonları ve çatışma durumunda kullanılacak çözüm yöntemleri anlatılacaktır.



1. HASH TABLOLARI

Doğrusal arama işleminin çalışma süresi $O(n)$ ile ifade edilebilir. İkili arama doğrusal aramadan daha hızlı çalışır ve $O(\log n)$ çalışma süresine sahiptir. İkili arama yöntemini kullanabilmek için listedeki elemanların sıralı olması gerekir. $O(n)$ ve $O(\log n)$ çalışma süresinden daha başarılı bir arama işlemi gerçekleştirebilmek amacıyla hash tabloları geliştirilmiştir. Hash tabloları kullanarak $O(1)$ çalışma süresi hedeflenmektedir. Bu yöntemin temeli, her bir aranan değer için sabit zamanda bir indis üretmek ve doğrudan tablodaki bu indisin gösterdiği alana erişmektir. Bir örnekle durumu açıklamak gerekirse, bir dersi alan 100 öğrencinin bilgilerini, boyutu 100 olan bir dizide tutmak mümkündür. Her bir öğrencinin dizide yer aldığı indis değeri öğrenci numarasından elde edilebilir. Örneğin öğrenci numarasının son iki hanesi, öğrencinin dizide yer aldığı alanı gösterebilir. Burada önemli olan verilen anahtar değerden hash tablosundaki bir alanı gösterecek bir indis üretebilmektir. Bu işlemi gerçekleştirmek için kullanılan fonksiyonlara hash fonksiyonları adı verilir. Bir hash fonksiyonunun bazı temel özelliklere sahip olması beklenir:

1. Hash fonksiyonlarının hesaplanması kolay olmalıdır.

2. Bu fonksiyonlar ile oluşan sürecin kararlı olması gerekir. Aynı girdi değerinin her zaman aynı çıktı değerini üretmesi beklenir. Tabloya eklenen bir değere daha sonra erişmek için aynı hash fonksiyonu kullanılacaktır.

3. Ayrıca bu fonksiyonların ürettiği değerlerin hash tablosu üzerindeki dağılımları dengeli olmalıdır. Bir başka ifade ile her bir hash tablosu indisinin eşit olasılıkla üretilmesi, bu fonksiyonun anahtar değerleri tablo üzerine yerleştirmesini kolaylaştıracaktır.

Hash tablolarında kullanılan fonksiyonlar için farklı yöntemler vardır. Bu yöntemler aşağıdaki gibi listelenebilir:

1. Bölme Yöntemi
2. Çarpma Yöntemi

3. Orta-Kare Yöntemi

4. Katlama Yöntemi

Bölme yönteminde kullanılan fonksiyon basit bir şekilde $\text{hash(anahtar)} = \text{anahtar} \bmod m$ şeklinde ifade edilebilir. Bu yöntemde anahtar değeri belirlenen bir m değerine bölünür ve kalan değer fonksiyonun ürettiği indis olarak değerlendirilir. m değeri tablonun boyutuna yakın bir asal sayı olarak belirlenir. Bu değerın asal olması anahtar değerlerin tablo üzerindeki dağılımının daha başarılı olmasını sağlar. Bu yöntemin en önemli dezavantajı birbirini takip eden anahtar değerlerin, hash tablosunda birbirini takip eden alanlara yerleştirilmesidir. Bu da hash tablosunun performansını düşürür. Ancak bu yöntemin avantajı anahtar değerleri farklı dizi alanlarına yerleştirmedeki başarısıdır. Örnek olarak boyutu 100 olan bir hash tablosunda m değerini 97 olarak seçelim. Anahtar değerın 62432 olduğunu varsayalım.

$\text{hash}(62432) = 62432 \bmod 97$ ifadesi indis olarak 61 değerini üretir.

Bir diğer yaklaşım olan çarpma yönteminde, 0 ile 1 arasında bir sabit değer seçilir. Anahtar değeri seçilen bu sabit değeri ile çarpılır ve elde edilen değerin kesirli kısmı alınır. Bu değeri tablonun boyutu ile çarpılarak tabloda kullanılacak indis değeri hesaplanır. k , 0 ile 1 arasında seçilen bir değeri olmak üzere, burada ifade edilen işlemlerin formül ile ifadesi aşağıdaki gibidir:

$$\text{hash(anahtar)} = \text{boyut} * (k * \text{anahtar} \bmod 1)$$

k için en uygun değeri 0.618 olduğu önerilmiştir.

5634 anahtar değerini boyutu 100 olan bir hash tablosuna çarpma yöntemi ile yerleştirelim.

$$\begin{aligned} \text{hash}(5634) &= 100 * (0.618 * 5634 \bmod 1) \\ &= 100 * (3481.812 \bmod 1) \\ &= 100 * 0.812 \\ &= 81 \end{aligned}$$

81, çarpma yönteminin 5634 anahtar değeri için ürettiği indistir.



ARAŞTIRALIM

Çarpma yönteminde kullanılan 0 ile 1 arasındaki sabit için 0.618 değerinin önerilme nedenini araştır-
alım.

Orta-kare yönteminde ise anahtar değerin karesi alınır. Elde edilen değerin ortada yer alan r hanesi hash fonksiyonun sonucu olarak değeri-
lendirilir. Anahtar değeri oluşturan tüm rakam-
lar r rakamı oluşturmada etki eder. Bu nedenle
bu yöntem de başarılı bir fonksiyon olarak düşü-
nülebilir. Bu yöntemde tüm anahtarlar için aynı
 r hane kullanılmalıdır.

Şimdi de orta-kare yöntemini kullanarak 6723
anahtar değerinin hash tablosundaki indisini he-
saplayalım. Sağdan 3. ve 4. hanelerin indis olarak
kullanılacağını düşünelim.

$$6723^2 = 45198729$$

Bu durumda 87 değeri 6723 anahtar değerinin
hash tablosundaki indis değeri olacaktır.

Son olarak katlama yönteminde ise anahtar
değer rakamsal olarak daha önceden belirlenmiş
uzunlukta parçalara ayrılır. Daha sonra bu parça-
lar toplanır. Her bir parçada yer alacak hane sayı-
sı hash tablosuna göre belirlenir. En son parçada
daha az hane bulunabilir.

32421 anahtar değerini boyutu 100 olan bir
hash tablosuna yerleştirmeye çalışalım. Değeri-
mizi 2 haneden oluşan sayılara parçalayalım. Bu
durumda elde edilen sayılar 32, 42 ve 1 olacaktır.
Buradaki örnekte de olduğu gibi son parça daha
az sayıda hane içerebilir. Bu rakamlar toplandı-
ğında 75 değeri indis değeri olarak üretilir.



ARAŞTIRALIM

anahtar mod m hash fonksiyonunda kullanılacak m değerinin asal sayı olmasının getirdiği avantaj-
ları araştıralım.



ARAŞTIRALIM

Zamanı veya benzer harici parametreleri kullanan hash fonksiyonlarını araştıralım.

Kullanılan hash fonksiyonunun yapısına bağ-
lı olarak, hash tablosuna yeni bir eleman ekleme
işleminde fonksiyon farklı anahtarlar için aynı
değeri üretebilir. Bir başka ifade ile farklı değer-
ler için tablo üzerinde kullanılacak aynı indis de-
ğerini hesaplayabilir. Oluşan bu duruma çatışma

(collision) denir. Tablodaki bir alan birden fazla
değer için kullanılamayacağı için bu probleme
bir çözüm üretmek zorunludur. Çatışma proble-
mini çözmek için temel olarak iki farklı yöntem
kullanılır.



DİKKAT EDELİM

Çatışma problemini mümkün olduğunca azaltmak için anahtar değerleri tablo üzerinde dengeli bir şekilde dağıtan hash fonksiyonlarını kullanalım.

2. AÇIK ADRESLEME (OPEN ADDRESSING)

Hash fonksiyonunun aşağıdaki gibi verildiğini varsayalım.

$$\text{hash(anahtar)} = \text{anahtar mod } 10$$

Anahtar değerlerimiz 1 ile 100 arasında verilsin. Aşağıda verilen değerleri hash tablosuna yerleştirmeye çalışalım.

82 27 35 47 64

Verilen bu anahtar değerlerden 82 yukarıda verilen fonksiyonun sonucu olarak tabloda 2 indisine, 27 7 indisine ve 35 5 indisine yerleştirilir. 47 değeri fonksiyona verildiğinde 7 indisi üretilir. Ancak 7 indisinde daha önce yerleştirilmiş 27 değeri vardır. Bu durumda çatışma meydana gelmiş olur. Bu durum Şekil 6.1' de gösterilmektedir.

0	
1	
2	82
3	
4	
5	35
6	
7	27 47 ?
8	
9	

Şekil 6.1. Hash tablosuna eklenme sürecinde çatışma durumunun oluşması

Çatışmaya sebep olan sorunu çözmek için kullanılan yöntemlerden birisi açık adreslemedir. Bu yöntemde belirli kurallar çerçevesinde çatışmaya sebep olan yeni anahtar tablonun farklı bir alanına yerleştirilir. Bu yerleştirme işlemini farklı şekillerde gerçekleştiren yöntemler vardır.

1. Doğrusal Sınama
2. Karesel Sınama
3. İkili Özetleme

Doğrusal sınama yönteminde çatışma olduğu anda bir sonraki boş alana bakılır. Bu işlem ilk boş alan bulununcaya kadar devam eder. Bir başka ifade ile hash değeri birer birer artırılır. Bu artırma işlemi aşağıdaki fonksiyon ile gösterilebilir.

$$\text{hash(anahtar)} = ((\text{anahtar mod } m) + i) \text{ mod } m$$

Bu formülde yer alan i değişkeni deneme sayısını göstermektedir. Bir anahtarı ilk defa hash tablosuna yerleştirmeye çalıştığımızda bu değişkenin değeri 0 olarak kabul edilir. İlk çatışma olduğunda i 1 olur. Aynı anahtar için her çatışmada i değeri 1 artırılır. Bu şekilde tablodaki ilk boş alana erişilinceye kadar artırma işlemi devam eder. Örneğin, 47 değeri eklenmek istendiği zaman ilk denemede 7 indisi üretilir. Bu alan hash tablosunda doludur. 0 olan i değeri 1 artırılır. Doğrusal sınama formülümüz kullanıldığında, yeni üretilen hash tablosu indisi 8 olur (Şekil 6.2).

0	
1	
2	82
3	
4	64
5	35
6	
7	27
8	47
9	

Şekil 6.2. 5 farklı değer doğrusal sına ile hash tablosuna eklenmesi

Karesel sına yönteminde çatışma olduğunda farklı bir formül ile boş alan bulunmaya çalışılır. Buradaki temel amaç kümeleme oluşmasını engellemektir. Çatışmadan sonra ilk boş alanı kullanmak yerine farklı bir değer hesaplanarak ilerlenir ve yerleştirme işlemi yapılır. Bu ilerleme miktarı, deneme sayısı ve bu sayının karesi daha önceden belirlenmiş sabit değerler ile çarpılarak elde edilir. Ancak yeni hesaplanan alanda da çatışma olursa deneme sayısını gösteren i değişkeni bir artırılır ve hesaplama işlemi tekrarlanır. Karesel sına yaklaşımında çatışma sonunda yeni yerleştirme alanı hesaplayan formül aşağıda verilmiştir.

$$\text{hash(anahtar)} = ((\text{anahtar mod } m) + s_1 i + s_2 i^2) \text{ mod } m$$

82 27 35 47 64

değerlerini karesel sına ile aynı hash tablosuna yerleştirelim. s_1 değerinin 1, s_2 değerinin 3 olarak belirlendiğini varsayalım. Hash fonksiyonu 47 değeri için ilk denemede ($i=0$) 7 değerini üretir. 7 indisine 27 anahtarı daha önce yerleştirildiği için 2. deneme ($i=1$) gerçekleştirilir. Fonksiyon, 47 anahtarı için 2. denemede 1 değeri üretir $((47 \text{ mod } 10) + 1*1 + 3*1) \text{ mod } 10$. Elde edilen yeni hash tablosu Şekil 6.3'te verilmiştir.

0	
1	47
2	82
3	
4	64
5	35
6	
7	27
8	
9	

Şekil 6.3. 5 farklı değer karesel sına ile hash tablosuna eklenmesi

İkili özetleme yönteminde 2 farklı hash fonksiyonu kullanılır.

$$\text{hash(anahtar)} = (\text{hash1(anahtar)} + i * \text{hash2(anahtar)}) \text{ mod } m$$

$$\text{hash1(anahtar)} = \text{anahtar mod } m$$

$$\text{hash2(anahtar)} = \text{anahtar mod } m-1$$

Deneme sayısını gösteren i değişkeni diğer yöntemlerde olduğu gibi ilk denemede 0 değerine sahiptir. Dolayısı ile ilk deneme için 2. hash fonksiyonunun bir etkisi yoktur. 2. denemeden

itibaren iki fonksiyonun birlikte ürettiği indis değeri kullanılarak hash tablosuna yerleştirme işlemi gerçekleştirilir.

82 27 35 47 64

değerlerinin ikili özetleme yaklaşımıyla boyutu 10 olan bir hash tablosuna yerleştirilme işlemi Şekil 6.4'te gösterilmiştir.

0	
1	
2	82
3	
4	64
5	35
6	
7	27
8	
9	47

Şekil 6.4. 5 farklı değer için ikili özetleme ile hash tablosuna eklenmesi

Tüm açık adresleme yöntemlerinde bazı anahtar değerler ilk denemede hash tablosuna yerleştirilemeyebilirler. Bu nedenle bir hash tablosunda bir anahtar değeri arama işleminde fonksiyonunun ürettiği ilk alanda değer bulunmazsa bu ilgili anahtar değer tablodaki olmadığı anlamına gelmez. Aynen tabloya ekleme işleminde olduğu gibi kullanılan fonksiyonda *i* (deneme sayısı) değişkeninin değerini bir artırarak arama işlemine devam etmek gerekir. Tabii ki bu süreç anahtar değeri buluncaya veya boş bir tablo alanına rastlayıncaya kadar devam eder. Burada dikkat edilmesi gereken en önemli nokta silme işleminde, silinen değer bulunduğ tablo

indisine farklı bir yapay değer yerleştirilerek bu alanda daha önce bir anahtar değeri olduğunu göstermektir. Bir başka ifade ile eğer silinme işleminde ilgili alan, bu alana hiç değer eklenmemiş gibi bir değer alır veya boş bırakılırsa bazı arama işlemlerinin tamamlanmadan kesilmesine yol açabilir. Arama işlemine devam edilebilmesi için herhangi bir denemede aradığımız anahtar değere sahip olmayan bir tablo alanının dolu olması gerekir. Silme işleminde, boş yerine “silindi” ifadesini temsil eden farklı bir sayısal değeri yerleştirmek, arama işlemine devam edilmesini sağlar.



DİKKAT EDELİM

Hash tablosundan bir anahtar değeri silindiğinde bu alanı boş alanlardan farklı bir ifade ile göstermeye dikkat edelim.

3. AYRIK ZİNCİRLEME (SEPARATE CHAINING)

Ayrık zincirleme yönteminde hash tablosunun her bir indisinde bir bağlı listenin adresi yer alır. Hash tablosuna ekleme işleminde anahtar değeri hash fonksiyonunun ürettiği alanda yer alan

bağlı listeye yerleştirilir. Eğer herhangi bir alana bir anahtar eklenmemişse o indisin değeri NULL olarak kalır. Bu yöntemde kümeleme problemi söz konusu olmaz. Anahtarların sayısının hash

tablosundaki alanlardan daha fazla olduğu durumlarda da rahatlıkla kullanılabilir. Anahtar sayısının artması ile birlikte hash tablosundaki bazı bağlı listelerde yer alan eleman sayısı artabilir ve bu durum performansın düşmesine sebep olabilir. Arama işleminde öncelikli olarak hash fonksiyonu kullanılarak tablonun ilgili alanı bulunur. Daha sonra bu alanın gösterdiği bağlı listede doğrusal arama gerçekleştirilir. En kötü durumda, eğer tüm anahtarlar (n tane anahtar olduğunu varsayalım) aynı bağlı listeye eklenmişse arama işleminin maliyeti $O(n)$ kadar olabilir. Doğru seçilmiş bir hash fonksiyonunun bu duruma izin vermesi mümkün değildir.

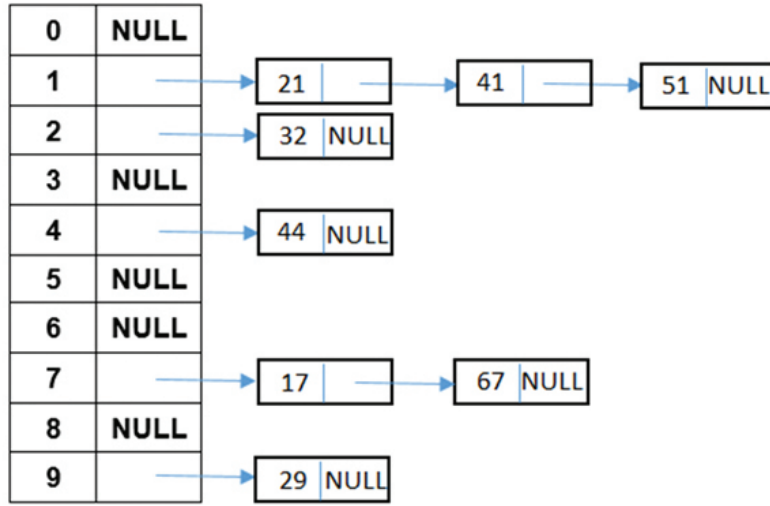
Hash fonksiyonunun aşağıdaki şekilde verildiğini düşünelim.

$$\text{hash(anahtar)} = \text{anahtar mod } m$$

Aşağıda verilen listeyi ayrı zincirleme kullanarak bir hash tablosuna ekleyelim.

32 44 21 17 41 29 67 51

Tüm değerler eklendiğinde Şekil 6.5'te yer alan hash tablosu elde edilir. Tabloda görüldüğü gibi kümeleme sorunu ortaya çıkmamıştır.



Şekil 6.5. 8 anahtarın boyutu 10 olan bir hash tablosuna ayrı zincirleme ile eklenmesi

BÖLÜM ÖZETİ

Bu bölümde bir veri yapısında temel olarak en fazla kullanılan ekleme ve arama işlemlerini $O(1)$ çalışma süresinde gerçekleştirmeyi hedefleyen hash tabloları anlatılmıştır. Bir anahtar değerinin bir hash tablosuna yerleştirilebilmesi için bu değerden tablonun boyutuna bağlı olarak geçerli bir indis değeri üretmek gerekir. Bu işlem için kullanılan farklı yöntemler ve fonksiyonlar söz konusudur. Bölme, çarpma, orta-kare ve katlama yöntemleri mevcuttur. En fazla kullanılan hash fonksiyonu bölme yöntemini temsil eden mod işleminin kullanıldığı fonksiyondur. Hash fonksiyonlarından beklenen kolay hesaplanabilme, kararlı olma ve anahtar değerleri tablo üzerinde dengeli bir şekilde dağıtma gibi özellikler vardır. Her ne kadar bir fonksiyon bu özelliklere de sahipse bazı durumlarda aynı tablo alanına birden fazla anahtarın hash değeri isabet edebilir. Bu durum çatışma olarak isimlendirilir ve bu duruma çözüm getirecek farklı yöntemler vardır. Açık adresleme yönteminde hash tablosundaki boş alanlar belirli bir düzen çerçevesinde çatışmaya neden olan anahtar değerler için kullanılır. Ayrık zincirleme yöntemi hash tablosunun indislerinin her birinde farklı bir bağlı listenin adresini tutar. Bu şekilde hash tablosunun boyutundan daha fazla sayıda anahtar değer bu veri yapısında yer almış olur.

GÖZDEN GEÇİRELİM

1. Aşağıdakilerden hangis hash tablolarında hedeflenen çalışma süresidir?

- a) $O(n^2)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n \log n)$
- e) $O(1)$

2. Aşağıdakilerden hangisi iyi bir hash fonksiyonunun özelliklerinden değildir?

- a) Tüm hash değerlerini yaklaşık olarak eşit ihtimalle üretmesi
- b) Aynı girdi değerinin aynı çıktı değerini üretmesi
- c) Bazı değer aralıklarında daha fazla çıktı vermesi
- d) Kolay hesaplanması
- e) Hash tablosunun dışında değer üretmemesi

3. Aşağıdakilerden hangisi anahtar mod m hash fonksiyonunda m in alabileceği en iyi değerdir?

- a) m değerinin asal olması
- b) m değerinin çift olması
- c) m değerinin tek olması
- d) m değerinin tablonun boyutu olması
- e) m değerinin 5'in katı olması

4. Aşağıdaki hash fonksiyonu üretme yöntemlerinden hangisi 0 ile 1 arasında bir sabit değer kullanır?

- a) Bölme yöntemi
- b) Çarpma yöntemi
- c) Orta-Kare yöntemi
- d) Katlama yöntemi
- e) Hiçbir fonksiyonda kullanılmaz

5.

0	
1	
2	72
3	
4	82
5	
6	
7	
8	
9	

Aşağıdakilerden hangisi doğrusal sına-
ma kullanılan yukarıdaki hash tablosu
için kesinlikle söylenemez (hash(anahtar) = anahtar mod 10)?

- a) Tabloda 2 anahtar değer yer almaktadır.
- b) Tabloya en az 3 değer eklenmiştir.
- c) Tabloya 52 değeri eklenmek istenirse 3 nolu indise eklenir.
- d) Tablodan herhangi bir değer silinmiştir.
- e) Tabloda yeni anahtar değerler için yer vardır.

6. Aşağıdaki hangi açık adresleme yönteminde deneme sayısı dışında, önceden belirlenen sabit değerler kullanılmaktadır?

- a) Katlama yöntemi
- b) Doğrusal sına-
ma
- c) Karesel sına-
ma
- d) İkili özetleme
- e) Ayrık zincirleme

7. Aşağıdakilerden hangisi, 43, 64 ve 33 anahtar değerlerinin doğrusal sına kullanılarak boyutu 5 olan bir hash tablosuna eklenmiş durumudur ($\text{hash(anahtar)} = \text{anahtar} \bmod 5$)?

a)

0	33
1	
2	
3	43
4	64

b)

0	64
1	
2	
3	33
4	43

c)

0	64
1	
2	
3	43
4	33

d)

0	
1	
2	33
3	43
4	64

e)

0	
1	33
2	
3	43
4	64

8.

0	64
---	----

1	51
2	62
3	43
4	21

Yukarıda verilen hash tablosunda yer alan değerlerin, doğrusal sına kullanılarak tabloya eklenme sırası aşağıdakilerden hangisi olabilir ($\text{hash(anahtar)} = \text{anahtar} \bmod 5$)?

a) 64 51 62 43 21

b) 51 43 62 21 64

c) 21 43 62 51 64

d) 51 43 62 64 21

e) 51 64 62 21 43

9. Doğrusal sına yönteminin en önemli dezavantajı aşağıdakilerden hangisidir?

a) Hesaplama zamanı

b) Kararlı bir yaklaşım olmaması

c) Kümeleme oluşmasına sebep olması

d) Çatışma sorununu çözmemesi

e) Bazı dizi indislerine çok fazla yerleştirme yapması

10. Aşağıdaki yöntemlerden hangisinde tabloya, tablonun boyutundan daha fazla sayıda anahtar yerleştirilebilir?

a) Doğrusal sına

b) Karesel sına

c) İkili özetleme

d) Ayrık zincirleme

e) Basit özetleme fonksiyonu

Yanıt Anahtarı: 1-E 2-C 3-A 4-B 5-D 6-C 7-A 8-B 9-C 10-D

Kaynakça

Deitel, P., Deitel, H. (2017). C++ How to Program. 10th Edition. Pearson.

Thareja R. (2014) Data Structures Using C, 2nd Edition, Oxford University Press.

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 7

ALGORİTMANIN ETKİNLİK ANALİZİ

ANAHTAR KAVRAMLAR

Etkinlik Analizi, Zaman Karmaşıklığı, Fonksiyon Büyüme Hızı, En – İyi Durum, En – Kötü Durum, Ortalama Durum, Asimptotik Notasyon

ÖĞRENME HEDEFLERİ

-  1 Algoritmanın etkinlik analizinin nasıl uygulanacağını bilir
-  2 Algoritmanın temel işleminin nasıl belirleneceğini bilir
-  3 Fonksiyonların büyüme hızının nasıl belirleneceğini bilir
-  4 Asimptotik notasyonları bilir
-  5 Asimptotik notasyonların algoritmaların etkinlik analizinde nasıl uygulanacağını bilir

GİRİŞ

Algoritmalar için iki tür etkinlik analizi göz önünde bulundurulmaktadır: zaman etkinliğinin analizi (ya da zaman karmaşıklığı) algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğinin belirlenmesi, alan etkinliğinin analizi (ya da alan karmaşıklığı) algoritmanın verilen girdi için istenen sonucu üretirken bellekten ne kadarlık kısma ihtiyaç duyduğu şeklinde tanımlanabilir. Etkinlik analizi algoritmanın pratikte uygulanıp uygulanamayacağına dair fikir verirken, aynı zamanda aynı problem için geliştirilmiş farklı algoritmaları da kıyaslayabilmemize imkan sağlamaktadır. Bu bölümde verilen bir algoritmanın etkinlik analizinin nasıl yapıldığı örnekler üzerinden detaylıca incelenecektir.



1. ÖLÇÜ BİRİMİNİN BELİRLENMESİ

Verilen bir algoritmanın zaman karmaşıklığını hesaplarırken, yani algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğini ölçerken; ilk yapılması gereken bu ölçümün hangi birim kullanılarak yapılacağını belirlenmesidir. Bu noktada akla ilk gelen zaman karmaşıklığını saniye ya da dakika gibi zaman birimleri cinsinden ölçmek olacaktır. Bunun için algoritmayı üzerinde çalıştırabileceğimiz bir bilgisayar bulmamız ve algoritmayı bu bilgisayarda çalıştırabileceğimiz bir programlama diline aktarmamız gerekmektedir. Sonrasında verilen bir girdi için algoritmayı çalıştırabilir ve bir kronometre yardımıyla algoritmanın kaç saniyede istenen çıktıyı ürettiğini belirleyebiliriz. Burada elbette daha iyi donanıma sahip bir bilgisayar ya da programla dili için daha etkin bir derleyici kullanmak algoritmanın çalışma süresini kısaltacaktır. Dolayısıyla; bu ölçme yönteminde bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı ya da programlama dili için hangi derleyicinin kullanıldığı gibi etkenler algoritmanın çalışma süresini yani zaman karmaşıklığını doğrudan etkilemektedirler.

Diğer yandan, geliştirdiğimiz algoritmaları günlük pratikte kullanabilmemiz için uygun programlama dillerine aktarmamız ve belli bir takım donanım cihazları satın almamız, yani zaman ve para harcamamız gerekmektedir. Dolayısıyla zamanımızı ve paramızı boşa harcamamak için, daha henüz kağıt üstündeyken geliştirdiğimiz algoritmaların etkinlik analizinin yapılması ve belki de yatırıma değer olup olmadıklarının belirlenmesi gerekmektedir. Ve yukarıda ifade edilenler düşünüldüğünde bu analizin, algoritmaların üzerinde çalıştırılacağı bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı gibi etkenlerden bağımsız bir şekilde yapılması daha doğru bir yöntem olacaktır.

Algoritmanın zaman karmaşıklığını ölçerken uygulayabileceğimiz bir yöntem de, algoritmanın istenen çıktıyı üretirken gerçekleştirdiği bütün işlemleri belirlemek ve bu işlemlerin algoritma sonlandırılıncaya kadar kaç defa gerçekleştirildiğini hesaplamak olabilir. Yalnız bu yöntemde algoritmanın gerçekleştirdiği bütün işlemleri belirlemek algoritmanın zaman karmaşıklığı hesabı için hem zor hem de gereksiz bir uğraş olacaktır. Onun yerine algoritmanın gerçekleştirdiği ve algoritmanın çalışma süresini doğrudan etkileyen en önemli ve temel işlemi belirlemek ve algoritma sonlandırılıncaya kadar bu işlemin kaç defa uygulandığını hesaplamak daha doğru bir yaklaşım olacaktır.

Şimdi bu yöntem üzerinden sözde kodu Şekil 7.1'deki gibi olan ve verilen bir tamsayı dizisinin verilen bir tamsayıyı içerip içermediğini bulmaya yarayan algoritmanın zaman karmaşıklığını hesaplamaya çalışalım. **Arama** algoritmasında **for** döngüsü içerisinde aranan tamsayının, dizinin sıradaki elemanına eşit olup olmadığını kontrol etmemize imkan veren ' $t_k = x$ ' kıyaslaması algoritmanın temel işlemidir. Örneğin [4, 1, 6, 3] tamsayı dizisi ve 10 tamsayısı verildiğinde, algoritma ' $t_k = x$ ' kod parçası yardımıyla sırasıyla dizinin elemanlarını 10 elemanı ile kıyaslayacak ve dizinin elemanları aranan tamsayıya eşit olmadığından '0' değerine dönecektir. Dikkat edilirse algoritma bu örneklem için 4 kıyaslama yapacaktır. Diğer yandan girdiyi [7, 21, 6, 5, 12, 8, 9, 17] tamsayı dizisi ve 11 tamsayısı olarak belirlersek; bu sefer algoritma girdi üzerinde 8 kıyaslama yapacak ve verilen tamsayı dizisi verilen 11 tamsayısını içermediğinden yukarıdaki örnekleme benzer şekilde '0' değerine dönecektir.



DİKKAT EDELİM

Algoritmanın etkinlik analizi algoritmaların üzerinde çalıştırılacağı bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı gibi etkenlerden bağımsız bir şekilde yapılmalıdır.

1	Arama ([$t_1, \dots, t_n$]; x)
2	
3	girdi : n boyutlu bir tamsayı dizi ve bir x tamsayısı
4	çıkıtı : dizi x 'i içeriyorsa, dizinin x 'e eşit ilk elemanının
5	indeksi; değilse, 0
6	
7	for ($k = 1$ to n)
8	if $t_k = x$
9	return k
10	return 0

Şekil 7.1 Arama Algoritmasının Sözde Kodu

Burada da görülebileceği gibi algoritmanın performansı ya da çalışma süresi ile algoritmanın girdi boyutu arasında doğrudan bir ilişki vardır. Daha büyük boyutlu dizilerin verilen bir elemanı içerip içermediğini belirlemek küçük boyutlu girdilere nazaran daha çok süre alacaktır. Dolayısıyla algoritmanın gerçekleştirdiği en temel işlemin algoritma tarafından kaç defa gerçekleştirildiğini hesap eden fonksiyon, algoritmanın girdi boyutu üzerinde tanımlanacaktır. İlgili fonksiyon girdi boyutu üzerinde tanımlandığından, etkinlik analizinde öncelikli olarak

girdi boyutunun belirlenmesi gerekmektedir. Örneğin yukarıdaki örnekte olduğu gibi verilen bir tamsayı dizisinin verilen bir tamsayıyı içerip içermediğini belirlerken girdi boyutu ilgili tamsayı dizisinin eleman sayısı iken, verilen iki matrisin çarpımında girdi boyutu ilgili matrislerin satır ve sütun sayıları olmaktadır. Verilen fonksiyonu $T(n)$ notasyonu ile gösterirsek; n elemanlı bir tamsayı dizisinin verilen bir x tamsayısını içerip içermediğini belirleyen ve Şekil 7.1'de sözde kod ile gösterimi verilen algoritmanın çalışma süresini veren fonksiyon $T(n) = n$ 'dir.

2. EN-İYİ DURUM, ORTALAMA DURUM VE EN-KÖTÜ DURUM ETKİNLİK ANALİZLERİ

Şekil 7.1'de verilen **Arama** algoritması yukarıda da gösterildiği gibi, [7, 21, 6, 5, 12, 8, 9, 17] tamsayı dizisi ve 11 tamsayısı için 8 kıyaslamada sonuçlanmakta ve '0' değerine dönmektedir. Halbuki algoritma aynı boyuttaki [12, 20, 13, 4, 11, 14, 9, 3] dizisi ve 12 tamsayısı için tek kıyaslamada sonuçlanacak ve '1' değerine dönecektir. Görüleceği üzere algoritmalar, aynı boyutta olsalar bile farklı yapıdaki girdiler için farklı zaman karmaşıklığına sahip olabilmekte ve farklı sayıda adımda sonuç üretebilmektedirler. Algoritmanın etkinlik analizi için bu farklı zaman karmaşıklıklarından hangisi dikkate alınacaktır? Gelin önce **Arama** algoritması üzerinden, verilen bir algoritma için bahsi geçen farklı etkinlik analizlerini belirleyelim.

[12, 20, 13, 4, 11, 14, 9, 3] dizisi ve 12 tamsayısı örnekleminde de gözlemlendiği üzere; eğer dizinin ilk elemanı aranan tamsayıya eşitse, girdi boyutu ne olursa olsun **Arama** algoritması

tek kıyaslamada sonuçlanmakta ve '1' değerine dönmektedir. Bu durum elde edilebilecek en iyi zaman karmaşıklığı olduğundan en-iyi durum etkinlik analizi olarak tanımlanmaktadır. Bu şekilde bir n boyutlu girdi için algoritmanın zaman karmaşıklığı $T(n) = 1$ olacaktır.

Diğer yandan [7, 21, 6, 5, 12, 8, 9, 17] dizisi ve 11 tamsayısı örnekleminde de gözlemlendiği üzere; eğer dizi aranan tamsayıyı içermiyorsa, **Arama** algoritması dizinin sonuna kadar dizinin elemanlarını sırayla aranan tamsayı ile kıyaslamakta ve n kıyaslama sonucunda '0' değerine dönmektedir. Bu durum bu algoritma için elde edilebilecek en kötü zaman karmaşıklığı olduğundan en-kötü durum etkinlik analizi olarak tanımlanmaktadır. Dolayısıyla bu şekilde bir n boyutlu girdi için algoritmanın zaman karmaşıklığı $T(n) = n$ olacaktır.

Bir diğer incelenmesi gereken etkinlik analizi, algoritmanın rastgele bir girdi üzerinde kaç

adımında sonuç ürettiğini belirleyen ve algoritmanın etkinlik analizi ile ilgili genel durumu ifade eden ortalama durum etkinlik analizidir. Yalnız ortalama durum etkinlik analizinin uygulanabilmesi için girdilerin farklı durumlarının olasılık dağılımlarının bilinmesi gerekmektedir. Ortalama durum analizinin nasıl uygulanacağını yine **Arama** algoritması üzerinden anlamaya çalışalım. Önce farklı durumlar için algoritmanın zaman karmaşıklıklarını belirleyelim. Aranan tamsayı, dizinin ilk elemanına eşit olduğunda zaman karmaşıklığı $T(n) = 1$ olduğunu daha önce belirlemiştik. Benzer şekilde aranan tamsayı dizinin ikinci elemanına eşitse algoritma iki kıyaslamada sonuçlanacağından, algoritmanın zaman karmaşıklığı $T(n) = 2$ olacaktır. Genelleyecek olursak; $i = 1 \dots n$ için, aranan tamsayı dizinin i . elemanına eşitse $T(n) = i$ olacaktır. Ayrıca dizi aranan sayıyı içermediğinde, algoritmanın zaman karmaşıklığının $T(n) = n$ olduğunu da daha önce belirlemiştik.

Algoritmanın yeterince girdi üstünde denendiğini varsayalım ve $i = 1 \dots n$ olmak üzere, bu girdilerden yüzde kaçında aranan sayının dizinin i . elemanına eşit olduğu veya yüzde kaçının aranan sayıyı içermediği belirlenmiş olsun. Bu durumda ortalama durum etkinlik analizini uygularsak; örneğin girdilerin tamamı aranan sayıyı içermiyorsa, algoritmanın zaman karmaşıklığı $T(n) = n$

olacaktır. Eğer girdilerin tamamı aranan sayıyı içeriyorsa ve $i = 1 \dots n$ için aranan sayının dizinin i . elemanına eşit olma durumlarının sayıları birbirine eşitse; algoritmanın zaman karmaşıklığı farklı durumların zaman karmaşıklıklarının ortalaması olacağından, $T(n) = (1 + 2 + \dots + n) / n$, yani $T(n) = (n + 1) / 2$ olacaktır.

Bu analizden anlaşılacağı üzere; her ne kadar ortalama durum etkinlik analizi algoritmanın performansı ile ilgili en doğru tespiti yapıyor olsa da, bu tür bir etkinlik analizi girdilerin farklı durumlarının olasılık dağılımlarının bilinmesini gerektirmektedir. Böyle bir dağılımı elde etmek veya farklı girdi durumları için verilen bir olasılık dağılımını doğrulamak da zordur. Dolayısıyla ortalama durum etkinlik analizini uygulamak en-kötü ve en-iyi durum etkinlik analizine nazaran daha çok çaba gerektirmektedir.

Diğer yandan; en-iyi durum etkinlik analizi, bu durumu oluşturacak girdilerin gelme olasılığı düşük olduğundan algoritmanın performansını değerlendirme noktasında çok da fikir veremeyecektir. Bununla birlikte, algoritmanın en uzun sürede sonuç ürettiği olası en kötü girdiler üzerinden etkinlik analizinin yapılması, performans açısından en kötü durumda bile algoritmanın nasıl çalıştığını gözlemlemeye imkan sağladığından, algoritmanın zaman karmaşıklığı için genellikle en-kötü durum analizi dikkate alınmaktadır.



DİKKAT EDELİM

Algoritmanın en uzun sürede sonuç ürettiği olası en kötü girdiler üzerinden etkinlik analizinin yapılması, performans açısından en kötü durumda bile algoritmanın nasıl çalıştığını gözlemlemeye imkan sağladığından, algoritmanın zaman karmaşıklığı için genellikle en-kötü durum analizi dikkate alınmaktadır.

3. FONKSİYONLARIN BÜYÜME HIZI

Yukarıda da ifade edildiği gibi, algoritmanın performansı ya da çalışma süresi ile algoritmanın girdi boyutu arasında doğrudan bir ilişki vardır. Girdi boyutu arttığında algoritmanın çalışma süresi de gözle görülür biçimde artmaktadır. Bu-

nunla birlikte girdi boyutu çok küçük olduğunda, çoğu algoritma için algoritmanın zaman karmaşıklığı da çok küçük olmaktadır. Dolayısıyla verilen bir algoritmanın etkinlik analizini yaparken çok küçük boyutlu girdiler kullanmak anlamlı

olmayacaktır. Yine benzer şekilde farklı algoritmaları etkinlikleri üzerinden kıyaslarken küçük boyutlu girdiler kullanıyor olmak ta sağlıklı sonuçlar üretmeyecektir.

Diğer yandan, algoritmaların etkinlik analiziyle asıl amaçlanan; algoritmaların tam olarak kaç adımda ya da tam olarak ne kadar sürede sonuç ürettiğini belirlemek değil, algoritmaların pratikte uygulanıp uygulanamayacağına dair bir fikir elde etmek ya da aynı problem için geliştirilmiş farklı algoritmaları performans açısından kıyaslayabilmektir. Bu tespitlerden hareketle; algoritmaların etkinlik analizinde yaptığımız aşlında, yukarıda tanımlanan ve algoritmanın temel işlemlerinin algoritma tarafından kaç defa gerçekleştirildiğini bulan $T(n)$ fonksiyonunun, küçük boyutlu girdiler için değil de büyük boyutlu girdiler için ne ürettiğini bulmak, veya en genel haliyle girdi boyutu n büyüdükçe $T(n)$ fonksiyonunun nasıl değiştiğini anlamaya çalışmak, yani $T(n)$ fonksiyonunun büyüme hızını belirlemek olacaktır. Örneğin $T(n) = 3n^3 + 2n^2 + 1$ fonksiyonunu göz önünde bulunduralım. $n = 10$ için $T(10) = 3.1000 + 2.100 + 1 = 3201$ olacaktır. Yine benzer şekilde $n = 100$ için $T(100) = 3.1000000 + 2.10000 + 1 = 3020001$ olacaktır. Görüleceği üzere n 'nin değeri büyüdükçe fonksiyonun büyümesine en çok katkı yapan terim $3n^3$ terimidir. Hatta bu noktada fonksiyonun büyümesine katkısı çok az olduğundan terimin kaysayısı 3 göz ardı edilirse, fonksiyonun büyümesine en çok katkı yapan n^3 terimidir. Dolayısıyla $T(n) = 3n^3 + 2n^2 + 1$ fonksiyonunun büyüme hızı n^3 'tür.

4. ASİMPOTİK NOTASYONLAR

Büyük – Oh (O) notasyonu asimptotik üst sınır olarak tanımlanmaktadır. Yani $O(f(n))$, $f(n)$ fonksiyonu ile aynı ya da daha küçük büyüme hızına sahip fonksiyonları içeren bir küme tanımlanmaktadır. Buna göre; örneğin $g(n) = 7n^2 + 2n - 5$ fonksiyonunun büyüme hızı n^2 olduğundan, $g(n) \in O(n^2)$ 'dir. Yine benzer şekilde, $h(n) = 3n + 2$ fonksiyonunun büyüme hızı n olduğundan ve n n^2 'den küçük olduğundan, $h(n) \in O(n^2)$ 'dir. Fakat, $r(n) = 2n^4 + n$ fonksiyonunun büyüme hızı n^4 olduğundan ve n^4 n^2 'den büyük olduğundan, $r(n) \notin O(n^2)$ 'dir.

Özetleyecek olursak, algoritma etkinlik analizinde önce algoritmanın gerçekleştirdiği temel işlem ya da işlemler belirlenecek ve sonrasında istenen sonucu üretmek için bu işlemlerin algoritma tarafından kaç defa uygulandığını hesap eden $T(n)$ fonksiyonu üretilecektir. Daha sonra ise, algoritmanın performansı için gösterge sayılabilecek $T(n)$ fonksiyonunun büyüme hızı belirlenecektir. $T(n)$ fonksiyonunun büyüme hızı bize algoritmanın pratikte uygulanıp uygulanamayacağına ya da algoritmanın uygulanabilmesi için gerekli olan yatırıma değip değmeyeceğine dair fikir verecektir. Aşağıda küçükten büyüğe sıralı algoritma etkinlik analizinde yaygın olarak kullanılan fonksiyon büyüme hızları verilmiştir:

$$1 \quad \log n \quad n \quad n \log n \quad n^2 \quad n^3 \quad 2^n \quad 3^n \quad n!$$

Yukarıdaki skalaya göre örneğin verilen bir problem için geliştirilmiş iki farklı algoritmanın etkinlik analizi neticesinde $T(n)$ fonksiyonlarının büyüme hızları sırasıyla $\log n$ ve n olarak belirlenmişse, birinci algoritmanın büyüme hızı daha düşük olduğundan, uygulamada performans açısından birinci algoritma tercih edilecektir. Algoritmaların etkinlik analizinde $T(n)$ fonksiyonlarının büyüme hızları gözetilerek farklı algoritmaları kıyaslamak ve sıralamak için ya da geliştirilen bir algoritmanın yukarıdaki büyüme hızları skalasındaki uygun yerini tespit edebilmek için yaygın olarak 3 farklı asimptotik notasyon kullanılmaktadır. Bunlar büyük – Oh (O), büyük – Omega (Ω) ve büyük – Theta (Θ) notasyonlarıdır.

Diğer yandan, büyük – Omega (Ω) notasyonu asimptotik alt sınır olarak tanımlanmaktadır. Yani $\Omega(f(n))$, $f(n)$ fonksiyonu ile aynı ya da daha büyük büyüme hızına sahip fonksiyonları içeren bir küme tanımlanmaktadır. Buna göre; örneğin $g(n) = 7n^2 + 2n - 5$ fonksiyonunun büyüme hızı n^2 olduğundan, $g(n) \in \Omega(n^2)$ 'dir. Yine benzer şekilde, $r(n) = 2n^4 + n$ fonksiyonunun büyüme hızı n^4 olduğundan ve n^4 n^2 'den büyük olduğundan, $r(n) \in \Omega(n^2)$ 'dir. Fakat, $h(n) = 3n + 2$ fonksiyonunun büyüme hızı n olduğundan ve n n^2 'den küçük olduğundan, $h(n) \notin \Omega(n^2)$ 'dir.

Son olarak, büyük – Theta (Θ) notasyonu asimptotik sıkı sınır olarak tanımlanmaktadır. Yani $\Theta(f(n))$, $f(n)$ fonksiyonu ile tam olarak aynı büyüme hızına sahip fonksiyonları içeren bir küme tanımlamaktadır. Buna göre; örneğin $g(n) = 7n^2 + 2n - 5$ fonksiyonunun büyüme hızı n^2 olduğundan, $g(n) \in \Theta(n^2)$ 'dir. Fakat, $h(n) = 3n + 2$ fonksiyonunun büyüme hızı n olduğundan ve n n^2 'den farklı olduğundan, $h(n) \notin \Theta(n^2)$ 'dir. Yine benzer şekilde, $r(n) = 2n^4 + n$ fonksiyonunun büyüme hızı n^4 olduğundan ve n^4 n^2 'den farklı olduğundan, $r(n) \notin \Theta(n^2)$ 'dir.

Asimptotik notasyonlar için $g(n) \in O(f(n))$ (ya da $g(n) \notin O(f(n))$) gösterimi, $g(n) = O(f(n))$ (ya da $g(n) \neq O(f(n))$) şeklinde de kullanılabilir. Ayrıca $g_1(n) = O(f_1(n))$ ve $g_2(n) = O(f_2(n))$ verildiğinde, $g_1(n) + g_2(n) = O(\max\{f_1(n), f_2(n)\})$ olacaktır. Yine benzer şekilde $g_1(n) = O(f_1(n))$ ve $g_2(n) = O(f_2(n))$ verildiğinde, $g_1(n).g_2(n) = O(f_1(n).f_2(n))$ olacaktır.

Yukarıda da ifade edildiği gibi; $O(f(n))$ $f(n)$ fonksiyonu ile aynı ya da daha küçük büyüme hızına sahip fonksiyonları içeren bir küme tanımladığından, örneğin $g(n) = 3n^2 + n + 2$ fonksiyonu $O(4n^3)$ kümesinin bir elemanı olabileceği

gibi, $O(2n^4)$ kümesinin de bir elemanı olabilmektedir. Yani $g(n) = 3n^2 + n + 2$ fonksiyonunun büyük – Oh gösterimi $g(n) = O(4n^3)$ olabileceği gibi, $g(n) = O(2n^4)$ de olabilmektedir. Bununla birlikte, $g(n) = 3n^2 + n + 2$ fonksiyonunun en sade ve en küçük büyük – Oh gösterimi $g(n) = O(n^2)$ 'dir. Dikkat edilirse, fonksiyonun en sade ve en küçük büyük – Oh gösterimi bize fonksiyonun büyüme hızını vermektedir.

Şimdi de $g(n) = (3n^3 + n)(2\log n + 1) + 5n^4$ fonksiyonunun en sade ve en küçük büyük – Oh gösterimini bulmaya çalışalım. Burada $g(n)$ fonksiyonu, $g_1(n) = 3n^3 + n$, $g_2(n) = 2\log n + 1$ ve $g_3(n) = 5n^4$ olmak üzere $g(n) = g_1(n).g_2(n).g_3(n)$ şeklinde yazılabilir. Önce $g_1(n)$, $g_2(n)$ ve $g_3(n)$ fonksiyonlarının büyük – Oh gösterimlerini bulalım. Görüleceği üzere, bu fonksiyonların en sade ve en küçük büyük – Oh gösterimleri sırasıyla $g_1(n) = O(n^3)$, $g_2(n) = O(\log n)$ ve $g_3(n) = O(n^4)$ 'tür. Yukarıda verilen $g_1(n) + g_2(n) = O(\max\{f_1(n), f_2(n)\})$ ve $g_1(n).g_2(n) = O(f_1(n).f_2(n))$ kurallarını kullanırsak $g(n)$ fonksiyonundaki $g_1(n).g_2(n)$ teriminin büyük – Oh gösterimi $g_1(n).g_2(n) = O(n^3\log n)$ olacak ve dolayısıyla $g(n)$ fonksiyonunun büyük – Oh gösterimi $g(n) = O(n^3\log n) + O(n^4) = O(n^4)$ olacaktır.

5. ÖZYİNELEMELİ OLMAYAN ALGORİTMALARIN ETKİNLİK ANALİZİ

Burada yukarıda tanımı verilen asimptotik büyük – Oh notasyonu kullanılarak, özyinelemeli olmayan algoritmaların zaman karmaşıklığının

nasıl bulunduğu örnekler üzerinden anlatılacaktır. Algoritmaların zaman karmaşıklığı için en-kötü durum etkinlik analizi uygulanacaktır.

1	Buyuk [$t_1, t_2, \dots, t_n$]]
2	
3	girdi : n tamsayıdan oluşan bir dizi
4	çıktı : dizinin en büyük elemanı
5	
6	temp $\leftarrow t_1$
7	for ($k = 2$ to n)
8	if $t_k > \text{temp}$
9	temp $\leftarrow t_k$
10	return temp

Şekil 7.2 Buyuk Algoritması Sözde Kodu

İlk olarak Şekil 7.2'de sözde kod ile gösterimi verilen ve verilen bir dizinin en büyük elemanını bulan algoritmayı göz önünde bulunduralım. Algoritma 'temp' isimli bir geçici değişken tanımlamakta ve dizinin ilk elemanını bu değişkene ata-

maktadır. Sonra **for** döngüsü yardımıyla sırasıyla ikinciden sonuncuya kadar dizinin elemanlarını 'temp' ile kıyaslamakta ve 'temp'in güncel değerinden daha büyük bir eleman keşfederse 'temp' değişkenine bu elemanı atamaktadır. Algoritma

döngünün sonunda 'temp' değişkeninin güncel değerine dönmektedir. Dikkat edilirse algoritmanın gerçekleştirdiği iki temel işlem vardır: kıyaslama ($t_i > \text{temp}$) ve eğer 'temp'in güncel değerinden daha büyük bir sayı keşfedilirse atama ($\text{temp} \leftarrow t_i$). Burada en-kötü durum etkinlik analizi için göz önünde bulundurulması gereken girdi elemanları küçükten büyüğe sıralı bir dizidir. Çünkü böyle bir girdi için; dizinin bir sonraki elemanı her zaman için 'temp' değişkeninin güncel değerinden büyük olacağından, algoritma

for döngüsünün her adımında hem kıyaslama hem de atama işlemini gerçekleştirecektir. for döngüsünün $n - 1$ adımı olduğundan, algoritma en-kötü durum için döngüde $2(n - 1)$ işlem gerçekleştirecektir. Algoritmanın başındaki 'temp' değişkenine başlangıç değerini atamayı da ayrı bir işlem olarak sayarsak $T(n) = 2(n - 1) + 1$ olacaktır. Hesaplanan $T(n) = 2n - 1$ fonksiyonunun en sade ve en küçük büyük - Oh gösterimi $T(n) = O(n)$ 'dir. Dolayısıyla Şekil 7.2'de verilen algoritmanın zaman karmaşıklığı $O(n)$ 'dir.

1	Adım(n)
2	
3	temp $\leftarrow$ 0
4	for (i = 1 to n)
5	for (k = i to n)
6	temp $\leftarrow$ temp + 1
7	return temp

Şekil 7.3 Adım Algoritmasının Sözde Kodu

Şimdi de benzer şekilde Şekil 7.3'de sözde kod ile gösterimi verilen algoritmayı göz önünde bulunduralım. Algoritma 'temp' isimli bir geçici değişken tanımlamakta ve 0 sayısını bu değişkene atamaktadır. Algoritma iç içe for döngüsünün her bir adımında 'temp $\leftarrow$ temp + 1' değişkeni üzerinden 'temp' değerini 1 artırmakta ve çıktı olarak 'temp' değişkeninin güncel değerine dönmektedir. Burada algoritmanın gerçekleştirdiği temel işlem atama işlemi 'temp $\leftarrow$ temp + 1'dir. İçteki for döngüsünün i'den n'ye kadar ($n - i + 1$)

adımı olduğundan, algoritma atama işlemini dıştaki for döngüsünün her bir adımında sırasıyla $n, n - 1, n - 2, \dots, 1$ defa gerçekleştirmektedir. Dolayısıyla algoritmanın başındaki 'temp' değişkenine başlangıç değerini atamayı da ayrı bir işlem olarak sayarsak $T(n) = n + n - 1 + n - 2 + \dots + 1 + 1 = (n^2 + n + 2) / 2$ olacaktır. Hesaplanan $T(n) = (n^2 + n + 2) / 2$ fonksiyonunun en sade ve en küçük büyük - Oh gösterimi $T(n) = O(n^2)$ 'dir. Dolayısıyla Şekil 7.3'de verilen algoritmanın zaman karmaşıklığı $O(n^2)$ 'dir.

1	Simetrik ($[t_{1,1}, \dots, t_{1,n}; \dots; t_{n,1}, \dots, t_{n,n}]$)
2	
3	girdi : nxn boyutlu bir matris
4	çıktı : matris simetrikse true; değilse false
5	
6	for (i = 1 to n)
7	for (k = 1 to n
8	if $t[i][k] \neq t[k][i]$
9	return false
10	return true

Şekil 7.4 Simetrik Algoritmasının Sözde Kodu

Son olarak Şekil 7.4'de sözde kod ile gösterimi verilen ve verilen bir matrisin simetrik olup olmadığını belirleyen algoritmayı göz önünde bulunduralım. Görüleceği üzere algoritmanın

temel işlemi kıyaslamadır ($t[i][k] \neq t[k][i]$). Dikkat edilirse, dıştaki for döngüsünde n adım olduğundan, ve döngünün her adımı için içteki for döngüsünde yine n defa kıyaslama gerçek-

leştirildiğinden, algoritma en-kötü durum için n^2 kıyaslama gerçekleştirecek ve dolayısıyla $T(n) = n^2$ olacaktır. Burada en-kötü durum etkinlik analizi için göz önünde bulundurulması gereken girdi simetrik bir matristir. Hesaplanan $T(n) =$

n^2 fonksiyonunun en sade ve en küçük büyük – Oh gösterimi $T(n) = O(n^2)$ 'dir. Dolayısıyla Şekil 7.4'de verilen algoritmanın zaman karmaşıklığı $O(n^2)$ 'dir.

1	Kuvvet(X, N)
2	
3	girdi : X tamsayısı ve N pozitif tamsayısı
4	çıktı : X'in N. kuvveti
5	
6	if N = 0
7	return 1
8	else
9	return X.Kuvvet(X, N – 1)

Şekil 7.5 Kuvvet Algoritmasının Sözde Kodu

6. ÖZYİNELEMELİ ALGORİTMALARIN ETKİNLİK ANALİZİ

Burada yukarıda tanımlı verilen asimptotik büyük – Oh notasyonu kullanılarak, özyinelemeli algoritmaların zaman karmaşıklığının nasıl bulunduğu örnekler üzerinden anlatılacaktır. Özyinelemeli algoritmalar için de, özyinelemeli olmayan algoritmaların etkinlik analizine benzer olarak; önce algoritmanın temel işlemleri belirlenecek ve algoritmanın bu temel işlemleri kaç defa uyguladığını hesaplayan $T(n)$ fonksiyonu üretilecektir. Yalnız özyinelemeli algoritmalar için $T(n)$ fonksiyonunu üretmek kolay olmamaktadır. Önce $T(n)$ fonksiyonu için özyineleme bağıntısının belirlenmesi ve sonrasında bu bağıntının çözülmesi gerekmektedir. $T(n)$ fonksiyonu için özyineleme bağıntısı belirlendikten ve çözüldükten sonra, algoritmanın zaman karmaşıklığı için yine yukarıdaki analizlere benzer $T(n)$ fonksiyonunun en sade ve en küçük büyük – Oh gösterimi bulunacaktır.

İlk olarak Şekil 7.5'de sözde kod ile gösterimi verilen ve verilen bir X tamsayısının N. kuvvetini bulan özyinelemeli algoritmayı göz önünde bulunduralım. Algoritma 'N = 0' koşulu sağlanırsa 1 sayısına dönmekte, aksi halde 'X.Kuvvet(X, N – 1)' komutuyla girdi boyutu 1 azaltılarak algoritma tekrar çalıştırılmaktadır. Algoritmanın gerçekleştirdiği temel işlemi kıyaslama olarak belirtirsek, algoritma içerisinde bir temel işlem gerçekleştir-

ilmekte ve girdi boyutu bir azaltılarak algoritma tekrar çalıştırılmaktadır. Algoritmanın zaman karmaşıklığına $T(n)$ dersek, algoritmanın içerisindeki özyineleme çağrısının (yani 'X.Kuvvet(X, N – 1)' komutunun) zaman karmaşıklığı $T(n – 1)$ olacağından, $T(n) = T(n – 1) + 1$ olacaktır. Girdi boyutu N = 0 olduğunda, algoritma tek işlem gerçekleştirerek 0 değerine döneceğinden $T(0) = 1$ 'dir. Şimdi $T(0) = 1$ başlangıç koşulunu göz önünde bulundurarak $T(n) = T(n – 1) + 1$ bağıntısını çözelim. Burada bağıntıyı çözmekten kasıt, eşitliğin sağ tarafının belli bir takım yöntemler kullanarak T fonksiyonuna bağımlılığını ortadan kaldırmak ve eşitliğin sağında sadece n değişkeninden oluşan ifadeler elde etmektir.

$$T(n) = T(n – 1) + 1 \quad (1)$$

ifadesinde n gördüğümüz yere n – 1 yazarsak $T(n – 1)$ 'in karşılığı $T(n – 1) = T(n – 2) + 1$ bulunacaktır. Bunu ilk bağıntıda eşitliğin sağındaki $T(n – 1)$ 'in yerine yazarsak,

$$T(n) = T(n – 1) + 1 = T(n – 2) + 1 + 1 \quad (2)$$

elde etmiş oluruz. Yine benzer şekilde (1) eşitliğinde n gördüğümüz yere n – 2 yazarsak $T(n –$

2)'in karşılığı $T(n - 2) = T(n - 3) + 1$ bulunacaktır. Bunu (2) bağıntısında eşitliğin sağındaki $T(n - 2)$ 'in yerine yazarsak,

$$T(n) = T(n - 2) + 1 + 1 = T(n - 3) + 1 + 1 + 1 \quad (3)$$

elde etmiş oluruz. Bu şekilde devam edilirse i . adımda

$$T(n) = T(n - i) + i.1 \quad (4)$$



DİKKAT EDELİM

Özyinelemeli algoritmaların etkinlik analizinde, önce temel işlemlerin algoritma tarafından kaç defa gerçekleştirildiğini hesap eden $T(n)$ fonksiyonu için özyineleme bağıntısı belirlenmeli ve sonrasında zaman karmaşıklığı için bu bağıntı çözülmelidir.

ifadesi elde edilecektir. Burada $i = n$ için (4) eşitliğinde $T(n) = T(0) + n$ elde edilecektir. $T(0) = 1$ olduğundan, (4) eşitliği $T(n) = n + 1$ olacaktır. Hesaplanan $T(n) = n + 1$ fonksiyonunun en sade ve

en küçük büyük - Oh gösterimi $T(n) = O(n)$ 'dir. Dolayısıyla Şekil 7.5'de verilen algoritmanın zaman karmaşıklığı $O(n)$ 'dir.

```

1  EnBuyuk([ $t_i, t_{i+1}, \dots, t_j$ ])
2
3  girdi : N elemanlı bir tam sayı dizisi
4  çıkı : Dizinin en büyük elemanı
5
6  // dizide tek bir eleman kaldıysa algoritma o elemana döner
7  if  $i = j$ 
8      return  $t_i$ 
9  else
10      $mid \leftarrow (i + j)/2$ 
11     // dizinin ilk yarısının en küçük elemanı
12      $buyuk1 \leftarrow \text{EnBuyuk}([t_i, \dots, t_{mid}])$ 
13     // dizinin ikinci yarısının en küçük elemanı
14      $buyuk2 \leftarrow \text{EnBuyuk}([t_{mid+1}, \dots, t_j])$ 
15     if  $buyuk1 < buyuk2$ 
16         return  $buyuk2$ 
17     else
18         return  $buyuk1$ 

```

Şekil 7.6 EnBuyuk Algoritmasının Sözde Kodu

Şimdi de Şekil 7.6'da sözde kod ile gösterimi verilen ve verilen bir tamsayı dizisinin en büyük elemanını bulan özyinelemeli algoritmayı göz önünde bulunduralım. ' $i = j$ ' koşulu sağlanırsa algoritma t_i sayısına dönmekte, aksi halde ' $\text{EnBuyuk}([t_i, \dots, t_{mid}])$ ' ve ' $\text{EnBuyuk}([t_{mid+1}, \dots, t_j])$ ' komutlarıyla girdi boyutu yarıya indirilerek algoritma tekrar çalıştırılmaktadır. Dikkat edilirse ' $mid \leftarrow (i + j)/2$ ' atamasını ve ' $buyuk1 < buyuk2$ '

kıyaslamasını da algoritmanın gerçekleştirdiği temel işlemlerden sayarsak, algoritma içerisinde 3 temel işlem gerçekleştirilmektedir. Ayrıca ' $\text{EnBuyuk}([t_i, \dots, t_{mid}])$ ' ve ' $\text{EnBuyuk}([t_{mid+1}, \dots, t_j])$ ' komutlarıyla algoritma iki defa daha çalıştırılmaktadır. Algoritmanın zaman karmaşıklığına $T(n)$ dersek, algoritmanın içerisindeki özyineleme çağrılarının (yani ' $\text{EnBuyuk}([t_i, \dots, t_{mid}])$ ' ve ' $\text{EnBuyuk}([t_{mid+1}, \dots, t_j])$ ' komutlarının) zaman karmaşıklığı

ğı $T(n/2)$ olacağından, $T(n) = 2T(n/2) + 3$ olacaktır. Girdi boyutu $N = 1$ olduğunda, algoritma tek işlem gerçekleştirerek ' t_i ' değerine döneceğinden $T(1) = 1$ 'dir. Şimdi $T(1) = 1$ başlangıç koşulunu göz önünde bulundurarak $T(n) = 2T(n/2) + 3$ bağıntısını çözelim. Yukarıda çözüme benzer şekilde,

$$T(n) = 2T(n/2) + 3 \quad (7)$$

ifadesinde n gördüğümüz yere $n/2$ yazarsak $T(n/2)$ 'nin karşılığı $T(n/2) = 2T(n/4) + 3$ bulunacaktır. Bunu ilk bağıntıda eşitliğin sağındaki $T(n/2)$ 'nin yerine yazarsak,

$$T(n) = 2T(n/2) + 3 = 2(2T(n/4) + 3) + 3 = 4T(n/4) + 3(1 + 2) \quad (8)$$

elde etmiş oluruz. Yine benzer şekilde (7) ifadesinde n gördüğümüz yere $n/4$ yazarsak $T(n/4)$ 'ün karşılığı $T(n/4) = 2T(n/8) + 3$ bulunacaktır. Bunu (8) bağıntısında eşitliğin sağındaki $T(n/4)$ 'ün yerine yazarsak,

$$T(n) = 4(2T(n/8) + 3) + 3(1 + 2) = 8T(n/8) + 3(1 + 2 + 4) \quad (9)$$

elde etmiş oluruz. Bu şekilde devam edilirse i . adımda

$$T(n) = 2^i T(n/2^i) + 3(1 + 2 + \dots + 2^{i-1}) \quad (10)$$

ifadesi elde edilecektir. Burada $i = \log n$ için $T(n) = 2^{\log n} T(1) + 3(1 + 2 + \dots + 2^{\log n - 1})$ olacaktır. Eşitliğin sağını düzenlersek, $T(1) = 1$ olduğundan $T(n) = 4n$ olacaktır. Hesaplanan $T(n) = 4n$ fonksiyonunun en sade ve en küçük büyük - Oh gösterimi $T(n) = O(n)$ 'dir. Dolayısıyla Şekil 7.6'da verilen algoritmanın zaman karmaşıklığı $O(n)$ 'dir.

BÖLÜM ÖZETİ

Bu bölümde algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğinin belirlenmesi olarak tanımlayabileceğimiz algoritmanın zaman karmaşıklığının nasıl hesaplandığı örnekler üzerinden detaylıca incelenmiştir. Bir algoritmanın verilen girdi için ne kadar sürede ya da ne kadar adımda istenen çıktıyı ürettiğini ölçerken; ilk yapılması gereken bu ölçümün hangi birim kullanılarak yapılacağına belirlenmesidir. Her ne kadar akla ilk gelen zaman karmaşıklığını saniye ya da dakika gibi zaman birimleri cinsinden ölçmek olsa da bu noktada bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı ya da programlama dili için hangi derleyicinin kullanıldığı gibi etkenler algoritmanın çalışma süresini yani zaman karmaşıklığını doğrudan etkilemektedirler. Algoritmaların zaman karmaşıklığını daha sağlıklı bir şekilde hesaplayabilmek için etkinlik analizinin, algoritmaların üzerinde çalıştırılacağı bilgisayarın sahip olduğu donanımın kalitesi, algoritmanın hangi programlama diline nasıl aktarıldığı gibi etkenlerden bağımsız bir şekilde yapılması daha doğru bir yöntem olacaktır. Algoritmanın zaman karmaşıklığını ölçerken uygulayabileceğimiz yöntem, algoritmanın istenen çıktıyı üretirken gerçekleştirdiği temel işlemleri belirlemek ve bu işlemlerin algoritma sonlandırılıncaya kadar kaç defa gerçekleştirildiğini hesaplayan bir $T(n)$ fonksiyonu tanımlamak olacaktır. Bu fonksiyon algoritmanın girdi boyutu üzerinde çalışacaktır. Sonrasında, algoritmanın performansı için gösterge sayılabilecek $T(n)$ fonksiyonunun büyüme hızı belirlenecektir. $T(n)$ fonksiyonunun büyüme hızı bize algoritmanın pratikte uygulanıp uygulanamayacağına ya da algoritmanın uygulanabilmesi için gerekli olan yatırıma değip değmeyeceğine dair fikir verecektir. Bu bölümde ayrıca etkinlik analizinde $T(n)$ fonksiyonlarının büyüme hızları gözetilerek farklı algoritmaları kıyaslamak ve sıralamak için ya da geliştirilen bir algoritmanın büyüme hızları skalasındaki uygun yerini tespit edebilmek için yaygın olarak kullanılan büyük – Oh (O), büyük – Omega (Ω) ve büyük – Theta (Θ) asimptotik notasyonları incelenmiştir. Diğer yandan asimptotik büyük – Oh notasyonu kullanılarak, özyinelemeli ya da özyinelemeli olmayan algoritmaların zaman karmaşıklığının nasıl bulunduğu da örnekler üzerinden anlatılmıştır.

GÖZDEN GEÇİRELİM

1. Toplam($t_1, \dots, t_n$)

```
temp ← 0
for (k = 1 to n){
    temp ← temp +  $t_k$ 
}
return temp
```

$T(n)$ fonksiyonu algoritmanın temel işlemi ya da işlemlerinin algoritma tarafından kaç defa gerçekleştirildiğini sağlayan fonksiyon olarak tanımlanmaktadır. Buna göre yukarıda sözde kod ile gösterimi verilen algoritmanın $T(n)$ fonksiyonunun kuralı aşağıdakilerden hangisinde doğru olarak verilmiştir?

- a) $T(n) = 2n^2 + 2$
- b) $T(n) = n^2 + 2$
- c) $T(n) = 2n + 1$
- d) $T(n) = n - 3$
- e) $T(n) = \log n + 3$

2. İkiliArama(A, i, k, x)

```
if (i >= k)
    return false
mid ← (i + k) / 2
if (A[mid] = x)
    return true
elseif (A[mid] > x)
    return İkiliArama(A, i, mid-1, x)
else
    return İkiliArama(A, mid+1, k, x)
```

Yukarıda sözde kod ile gösterimi verilen özyinelemeli algoritmanın zaman karmaşıklığının özyineleme bağıntısı aşağıdakilerden hangisinde doğru olarak verilmiştir?

- a) $T(n) = T(n/2) + n$
- b) $T(n) = T(n/2) + 4$
- c) $T(n) = 2T(n/2) + 1$
- d) $T(n) = 3T(n/2) + 2$
- e) $T(n) = 4T(n/2) + 3$

3. Büyük – Omega (Ω) notasyonu asimptotik alt sınır olarak tanımlanmaktadır. Buna göre $f(n) = 2n^3 + 1$ fonksiyonu için aşağıdakilerden hangisi doğru değildir?

- a) $f(n) = \Omega(\log n)$
- b) $f(n) = \Omega(n)$
- c) $f(n) = \Omega(n^2)$
- d) $f(n) = \Omega(n^3)$
- e) $f(n) = \Omega(n^4)$

4. Büyük – Theta (Θ) notasyonu asimptotik sıkı sınır olarak tanımlanmaktadır. Buna göre $f(n) = n^2 + 3$ fonksiyonu için aşağıdakilerden hangisi doğrudur?

- a) $f(n) = \Theta(\log n)$
- b) $f(n) = \Theta(n)$
- c) $f(n) = \Theta(n^2)$
- d) $f(n) = \Theta(n^3)$
- e) $f(n) = \Theta(n^4)$

5. Büyük – Oh (O) notasyonu asimptotik üst sınır olarak tanımlanmaktadır. Buna göre $f(n) = 2n^3 + 1$ fonksiyonu için aşağıdakilerden hangisi doğru değildir?

- a) $f(n) = O(n)$
- b) $f(n) = O(n^3)$
- c) $f(n) = O(n^3 \log n)$
- d) $f(n) = O(n^4)$
- e) $f(n) = O(n^5)$

6. Kuvvet(X,n)

```
temp ← 1
for ( k = 1 to n){
    temp ← temp.x}
return temp
```

Yukarıda sözde kod ile gösterimi verilen algoritmanın zaman karmaşıklığı aşağıdakilerden hangisinde doğru olarak verilmiştir?

- a) $O(\log n)$
- b) $O(n)$
- c) $O(n \log n)$
- d) $O(n^2)$
- e) $O(n^3)$

7. Toplam([t₁..., t_n])

```
temp1 ← 0
for ( i = 1 to n){
    for ( j = 1 to n){
        temp2 ← 0
        for ( k = 1 to n){
            temp2 ← temp2 + k
            temp1 ← i.temp1 - j.temp2}}
return temp1
```

Yukarıda sözde kod ile gösterimi verilen algoritmanın zaman karmaşıklığı aşağıdakilerden hangisinde doğru olarak verilmiştir?

- a) $O(\log n)$
- b) $O(n)$
- c) $O(n \log n)$
- d) $O(n^2)$
- e) $O(n^3)$

8. EnBuyuk([t₁..., t_n])

```
temp ← t1
for ( k = 2 to n){
    if (tk > temp){
        temp ← tk}}
```

Yukarıda sözde kod ile gösterimi verilen EnBuyuk algoritmasının farklı elemanlardan oluşan bir sayı dizisini girdi olarak aldığı kabul edilirse en-iyi durum zaman karmaşıklığını verecek olan girdi aşağıdakilerden hangisidir?

- a) Girdinin ilk elemanının en büyük eleman olması
- b) Girdinin son elemanının en büyük eleman olması
- c) Girdinin küçükten büyüğe sıralı elemanlardan oluşması
- d) Girdinin ortanca elemanının en büyük eleman olması
- e) Girdinin ilk yarısının küçükten büyüğe ikinci yarısının büyüktan küçüğe elemanlardan oluşması

9. Büyük – Oh (O) notasyonu asimptotik üst sınır olarak tanımlanmaktadır. Buna göre $f(n) = n^7 + 3$ fonksiyonu için aşağıdakilerden hangisi doğrudur?

- a) $f(n) = O(n)$
- b) $f(n) = O(n^2)$
- c) $f(n) = O(n^4)$
- d) $f(n) = O(n^6)$
- e) $f(n) = O(n^8)$

10. Büyük – Omega (Ω) notasyonu asimp-totik alt sınır olarak tanımlanmaktadır. Buna göre $f(n) = n^2+3$ fonksiyonu için aşağıdakilerden hangisi doğrudur?

- a) $f(n) = \Omega(n)$
- b) $f(n) = \Omega(n^3)$
- c) $f(n) = \Omega(n^5)$
- d) $f(n) = \Omega(n^7)$
- e) $f(n) = \Omega(n^9)$

Yanıt Anahtarı: 1-C 2-B 3-E 4-C 5-A 6-B 7-E 8-A 9-E 10-A

Kaynakça

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 8

ARAMA ALGORİTMALARI

ANAHTAR KAVRAMLAR

Doğrusal Arama, İkili Arama,
Enterpolasyon Arama, Atlamalı Arama

ÖĞRENME HEDEFLERİ

-  1 Arama problemini tanımlar, başarılı ve başarısız arama kavramlarını bilir
-  2 Doğrusal arama algoritmasını uygulayabilir
-  3 İkili arama yöntemini kullanabilir
-  4 Enterpolasyon arama algoritmasını kullanabilir
-  5 Atlama arama yöntemini uygulayabilir

GİRİŞ

Bir değerin belirli bir liste içerisinde yer alıp almadığını belirlemek için gerçekleştirilen işleme arama işlemi denir. Arama işleminin sonunda, verilen değer liste içinde bulunmuş ise arama işlemi başarılı olarak kabul edilir. Arama işleminin kullanıldığı problem veya uygulamaya bağlı olarak başarılı bir aramada, aranan değerin liste içindeki adresi çevrilir. Bu değer listeyi oluşturmak için dizi kullanılmış ise, dizinin indisi, bağlı liste kullanılmış ise aranan elemanın yer aldığı düğümün adresidir. Arama işleminde istenen değer liste içinde yer almıyor ise arama işlemi başarısız olarak kabul edilir. Bu durumu göstermek için dizi kullanılıyorsa -1, bağlı liste kullanılıyorsa NULL değeri çevrilir. Arama işlemi için farklı yöntemler kullanılabilir. Doğrusal arama ve ikili arama yöntemleri en bilinen yöntemlerdir. İkili arama yönteminde listedeki elemanların sıralı olması gerekir. Listedeki elemanların sıralı olması halinde kullanılacak diğer yaklaşımlar enterpolasyon arama ve atlamalı arama algoritmalarıdır.



1. ARAMA ALGORİTMALARI

Programlama problemlerinin çözümünde farklı veri yapıları kullanılabilir. Kullanılan veri yapıları ile uygulamalarda yer alan bilgilerin verimli bir şekilde işlenmesi hedeflenir. Problemin veya uygulamanın yapısına göre farklı veri tiplerinden tanımlanmış elemanlar ilgili veri yapılarına yerleştirilirler. Bazı problemlerde verilen bir elemanın bir veri yapısı içinde yer alıp almadığının belirlenmesi gerekir. Bu problem arama problemi olarak tanımlanır. Arama problemi genel bir yaklaşımla verilen bir liste içerisinde aranan bir değer var olup olmadığının araştırılması olarak da ifade edilebilir. Arama işleminde, aranan değer liste içinde bulunursa, değer listesi içindeki adresi çevrilir. Bu adres kullanılan veri tipine göre farklılık gösterir. Eğer bir dizi kullanı-

lıyor ise bir indis değeri, bağlı liste kullanılmış ise adres değeri döndürülür. Eğer aranan değer liste içinde bulunamamışsa arama işleminin başarısız olduğunu gösteren -1 veya benzeri bir indis değeri veya NULL adres değeri çevrilir. Listede yer alan elemanların sıralı olup olmamasına bağlı olarak farklı arama algoritmaları kullanılabilir.

Bu bölümde arama algoritmalarını anlatmak için listedeki elemanların bir dizi içinde yer aldığı düşünülmüştür. Tüm arama fonksiyonları bir diziyi ve aranacak değeri parametre olarak alır. Fonksiyonlar, bu parametrelere ek olarak farklı değerler de kullanabilirler. Örneğin ikili arama fonksiyonu çağrılırken dizinin ilk ve son elemanının indis değerleri, diğer fonksiyonlar çağrılırken ek olarak dizinin boyutu kullanılmıştır.



DİKKAT EDELİM

Listedeki elemanların sıralı olup olmamasına göre farklı arama algoritmaları uygulanabilir



ARAŞTIRALIM

Arama algoritmalarını, listedeki elemanların bağlı listede yer aldığı problemler üzerinde uygulanmasını araştıralım.

2. DOĞRUSAL ARAMA

Arama algoritmaları arasında en basit olan yaklaşımdır. Bu yöntemi uygulamak için listedeki elemanların sıralı düzende olması gerekli değildir. Listede yer alan ilk elemandan başlayarak aranan değer sırasıyla her eleman ile karşılaştırılır. Aranan değer liste içinde bulunursa bu değer indisi çevrilerek arama işlemi başarılı bir şekilde sona erer. Tüm liste elemanları içerisinde aranan değer bulunamaması durumunda, bu durumu ifade etmek amacı ile -1 değeri çevrilir. Şekil 8.1'de verilen linearSearch fonksiyonu doğrusal arama yöntemini gerçekleştirmektedir. Bu fonksiyon, listenin yer aldığı diziyi, dizideki

eleman sayısını ve aranan değeri parametre olarak almaktadır. Aranan değer, listede yer alan elemanlar ile ilk elemandan başlayarak karşılaştırılır. Eğer değer bulunursa, elemanın yer aldığı indis değeri çevrilir. Tüm elemanlar ile karşılaştırma sonrasında değer bulunamazsa fonksiyon tanımında görüldü gibi -1 değeri çevrilmektedir. Şekil 8.2'de verilen örnekte 48 değeri liste içinde aranıyorsa, arama işleminin sonucu olarak 7 indisi çevrilir. Eğer aynı örnek üzerinde 47 değeri aranan değer ise, bu değer listede olmadığı için tüm elemanlar ile karşılaştırma işlemi sonucunda -1 değeri cevap olarak döndürülür.

```

1  int linearSearch(int ar[], int n, int value)
2  {
3      int i;
4      for (i = 0; i < n; i++) {
5          if (ar[i] == value)
6              return i;
7      }
8      return -1;
9  }

```

Şekil 8.1. Doğrusal arama fonksiyonu

0	34
1	45
2	27
3	11
4	93
5	46
6	25
7	48
8	13
9	21

Şekil 8.2. Doğrusal arama yönteminin örnek bir listede uygulanması

3. İKİLİ ARAMA

İkili arama yöntemini uygulayabilmek için listede yer alan elemanların sıralı bir düzene sahip olması gerekir. Listede yer alan elemanların artan veya azalan sırada olması durumunda ikili arama algoritması kullanılabilir. Bu bölümde verilecek ikili arama fonksiyonu listenin artan sırada olduğunu varsaymaktadır. Azalan sıralı listelerde, verilen bu fonksiyon, karşılaştırma işlemi basit bir şekilde değiştirilerek uygulanabilir. İkili arama fonksiyonu Şekil 8.3'te verilmektedir. Bu yöntemde listede yer alan en düşük indis ile en yüksek indis değerlerinin ortalaması bulunur. Bu ortalama indis değerinde yer alan eleman ile aranan değer karşılaştırılır. Eğer bir eşitlik söz konusu ise aranan değer bulunmuş olur ve ilgili indis değeri çevrilir. Eğer aranan değer bu elemandan daha büyükse, listenin ilk yarısı elenir. En küçük indis ortalama indis değerinin bir fazlası olarak güncellenir. Eğer aranan değer ortalama indiste

yer alan elemandan daha küçükse listenin ikinci yarısı elenir. En büyük indis gösteren değer ortalama indis değerinin bir eksiği olarak güncellenir. Bu güncellemelerden sonra eğer küçük indis değeri büyük indis değerinden küçük veya eşit olma durumunu sürdürüyorsa ortalama indis değeri tekrar hesaplanır ve ilgili karşılaştırmalar tekrarlanır. Eğer küçük indis gösteren indis değerinin büyük indis gösteren değerden büyük olma durumu söz konusu olursa aranan değer listede olmadığı anlaşılır ve -1 indis değeri çevrilir. Şekil 8.4'te verilen örnekte 15 elemanlı sıralı bir liste yer almaktadır. 18 değerinin arandığını varsayalım. İlk hesaplanan indis değeri 7'dir. Aranan 18 değeri, hesaplanan indiste yer alan 52 değerinden küçüktür. Bu nedenle listenin ikinci yarısı elenir. En büyük indis değeri ilk hesaplanan indis değerinin 1 eksiği olan 6 olarak belirlenir. Arama işlemi listenin ilk yarısında devam

eder. Yeni hesaplanan indis 3 tür ve indiste yer alan değer 22'dir. Aranan 18 değeri 22 den küçük-tür bu nedenle tekrar listenin 2. yarısı elenir. En büyük indis değeri 3 ün bir eksiği alan 2 olarak değiştirilir. Arama işleminin devam edeceği alt

listede en küçük indis değeri 0 ve en büyük indis değeri 2'dir. Buradan hesaplanan yeni ortalama indis 1 olur. Bu indiste yer alan değer aranan 18 değerine eşittir ve 1 indis değeri cevap olarak çevrilir.

```

1  int binarySearch(int ar[], int low, int high, int value)
2  {
3      while (low <= high) {
4          int m = (low + high) / 2;
5          if (ar[m] == value)
6              return m;
7          if (ar[m] < value)
8              low = m + 1;
9          else
10             high = m - 1;
11     }
12     return -1;
13 }
```

Şekil 8.3. İkili arama fonksiyonu

0	15
1	18
2	21
3	22
4	37
5	40
6	48
7	52
8	55
9	67
10	75
11	77
12	79
13	83
14	86

Şekil 8.4. İkili arama yönteminin örnek bir listede uygulanması



ARAŞTIRALIM

İkili arama algoritmasının, listenin boyutu artarken doğrusal arama algoritmasına karşı üstünlüğünü araştıralım.

4. ENTERPOLASYON ARAMA

Enterpolasyon arama kavramı, bir telefon rehberindeki isimleri arama yöntemine benzer. Örneğin, bir telefon rehberinde Burak ismini ararken, onun rehberin ilk kısımlarında yer alacağı tahmin edilebilir. Bu nedenle listeyi her seferinde ikiye bölerek ikili arama yaklaşımın kullanmak, bu problem için doğru bir yöntem değildir. Bu ve benzeri problemlerde aranan elemanın değeri arama işlemine avantaj oluşturmak için kullanılabilir. Enterpolasyon aramada, ikili arama yönteminde olduğu gibi listenin sıralı olması gerekir. Listenin ilk elemanı ve son elemanı kullanılarak, aranan değerinin listedeki yeri konusunda yaklaşık bir indis değeri üretilir. Bu değer hesaplanması Şekil 8.5'te yer alan interpolationSearch fonksiyonunda 11. satırda verilmiştir. Eğer üretilen

bu indis değerinde yer alan değer aranan değere eşitse arama işlemi başarılı olarak tamamlanmış olur ve indis değeri çevrilir. Eğer hesaplanan indis değerinde yer alan değer aranan değerden küçükse listenin ilk elemanını gösteren indis değeri hesaplanan indis değerinin bir fazlası olarak belirlenir. Hesaplanan indis değerinin gösterdiği değer aranan değerden büyükse listenin son elemanını gösteren indis değeri hesaplanan indis değerinin bir eksiği olarak atanır. Bu şekilde arama işleminin gerçekleştiği aralık daraltılır. Enterpolasyon arama yöntemini uygulayan, interpolationSearch fonksiyonu listenin yer aldığı diziyi, listenin eleman sayısını ve aranan değeri parametre olarak almaktadır.

```

1      int interpolationSearch(int ar[], int n, int value)
2      {
3          int low = 0, high = (n - 1);
4          while (low <= high && value >= ar[low] && value <= ar[high]) {
5              if (low == high)
6              {
7                  if (ar[low] == value) return low;
8                  return -1;
9              }
10             int pos;
11             pos = low + (((double)(high - low) / (ar[high] - ar[low])) * (value - ar[low]));
12
13             if (ar[pos] == value)
14                 return pos;
15
16             if (ar[pos] < value)
17                 low = pos + 1;
18             else
19                 high = pos - 1;
20         }
21         return -1;
22     }

```

Şekil 8.5. Enterpolasyon arama fonksiyonu



ARAŞTIRALIM

Sıralı listelerde kullanılabilecek farklı arama algoritmalarını araştırılın.

5. ATLAMALI ARAMA

Listemizin sıralı olması durumunda kullanabileceğimiz bir diğer arama yöntemi atlamalı arama yöntemidir. Bu yöntemde listede yer alan tüm elemanlar taranmak zorunda değildir. Listede yer alan bir eleman aranan değer ile karşılaştırılır. Karşılaştırılan bu eleman aranan değerden küçükse belirli sayıda eleman atlanarak farklı bir eleman ile karşılaştırma gerçekleştirilir. Karşılaştırılan yeni eleman aranan değerden büyükse, aranan değerın yer alabileceği bir sınır elde edilmiş olur. Aranan değerın, bir önce karşılaştırılan değer ile son karşılaştırılan değer arasında olduğu anlaşılmış olur. Karşılaştırılan değerın daha küçük olması durumunda yeniden belirli sayıda eleman atlanarak işlem tekrar edilir. Atlama miktarı listedeki eleman sayısına bağlı olarak ve genellikle eleman sayısının karekökü olarak hesaplanır. Aranan değerın yer aldığı aralık bulunduğunda bu aralık içerisinde doğrusal arama gerçekleştire-

rek arama işlemi tamamlanır. Şekil 8.6'da atlama-
lı arama yöntemini uygulayan jumpSearch fonksiyonu verilmektedir. Şekil 8.7'de yer alan sıralı listede 49 değerini arayalım. Şekil 8.6'da verilen jumpSearch fonksiyonunda görüldüğü gibi atlama miktarı listedeki eleman sayısının karekökü olarak belirlenir. Bu örnekte eleman sayısı 10 olarak verilmiştir ve atlama miktarı 3 olarak hesaplanır. Aranan 49 değeri öncelikli olarak 3. eleman 29 ile karşılaştırılır. 29 aranan değerden küçük olduğu için, bir sonraki karşılaştırma işlemi atlama miktarı kadar eleman sonrasında 6. eleman olan 41 ile gerçekleştirilir. 41'de aranan değer 49'dan küçüktür ve karşılaştırma sırası 9. eleman olan 55 değerine gelir. 55 aranan değer 49'dan büyüktür ve aranan değerın yer aldığı aralık belirlenmiş olur. Bu aralık içerisinde doğrusal arama işlemi ile 49 değerinin yer aldığı indis değeri bulunur ve 7 cevap olarak döndürülür.

```

1      int jumpSearch(int ar[], int n, int value)
2      {
3          int step = sqrt(n);
4          int prev = 0;
5          while (ar[min(step, n)-1] < value)
6          {
7              prev = step;
8              step += sqrt(n);
9              if (prev >= n)
10                 return -1;
11          }
12          while (ar[prev] < value)
13          {
14              prev++;
15              if (prev == min(step, n))
16                 return -1;
17          }
18          if (ar[prev] == value)
19              return prev;
20
21          return -1;
22      }

```

Şekil 8.6. Atlama arama fonksiyonu

0	12
1	25
2	29
3	33
4	37
5	41
6	45
7	49
8	55
9	64

Şekil 8.7. Atlamalı arama yönteminin örnek bir listede uygulanması



ARAŞTIRALIM

Atlamalı arama algoritmasında kullanılan atlama miktarını farklı şekilde hesaplayan yaklaşımları araştırılım.



DİKKAT EDELİM

İkili arama, enterpolasyon arama ve atlamalı arama algoritmaları sadece sıralı listelerde kullanılabilir.

BÖLÜM ÖZETİ

Bu bölümde temel arama algoritmaları anlatılmıştır. Verilen liste içinde bir değerin yer alıp almadığının araştırılması işlemi olarak tanımlanan arama problemi, listenin sıralı olup olmamasına bağlı olarak farklı yöntemler ile çözülebilir. Liste sıralı değil ise temel yaklaşım ilk elemandan başlayarak tüm elemanların aranan değere eşit olup olmadığını kontrol etmek olacaktır. Doğrusal arama olarak isimlendirilen bu çözüm bir anlamda kaba kuvvet algoritmasıdır. Eğer verilen liste sıralı ise ikili arama, enterpolasyon arama ve atlamalı arama gibi farklı çözümler kullanılabilir. İkili arama $O(\log n)$ çalışma zamanında arama işlemini tamamlamayı hedefleyen bir algoritmadır. Her karşılaştırma sonucunda listenin yarısı elenir. Enterpolasyon arama yönteminde, sıralı listenin ilk ve son elemanlarının değerleri kullanılarak aranan değerin liste içindeki yeri tahmin edilmeye çalışılır. Bu arama yönteminde aranan elemanın sayısal değerinin, elemanın listenin hangi bölgesinde yer alacağına yönelik bir avantaj üretmesi beklenir. Tabii ki bu tahminde listenin ilk ve son elemanlarının değerleri de büyük önem taşımaktadır. Aranan değer bulunamazsa değerin hesaplanan indiste yer alan değer den büyük veya küçük olmasına bağlı olarak oluşturulan alt listede aynı yöntemle arama işlemi devam eder. Sıralı listelerde kullanılan diğer bir yöntem atlamalı arama algoritmasıdır. Bu yöntemde aranan değer ile karşılaştırma işlemi gerçekleştirilecek elemanlar bir atlama değeri kullanılarak belirlenir. Atlama miktarı eleman sayısının karekökü hesaplanarak elde edilir. Bu karşılaştırmalar sonucunda aranan değerin yer aldığı aralık belirlenir. Bu aralık içinde doğrusal arama ile aranan değerin yer aldığı indis belirlenir.

GÖZDEN GEÇİRELİM

1. Aşağıdaki arama algoritmalarından hangisinde listedeki elemanların sıralı olması gerekmez?
 - a) Doğrusal arama
 - b) İkili arama
 - c) Enterpolasyon arama
 - d) Atlamalı arama
 - e) Hiçbiri
2. Aşağıdaki arama algoritmalarından hangisinde karşılaştırma işleminde, eğer aranan değer bulunamazsa listedeki elemanların yarısı elenir?
 - a) Enterpolasyon arama
 - b) Hepsi
 - c) İkili arama
 - d) Doğrusal arama
 - e) Atlamalı arama
3. Atlamalı arama algoritmasında atlama miktarı aşağıdaki yöntemlerden hangisi ile hesaplanabilir?
 - a) Listedeki eleman sayısının yarısı
 - b) Listedeki eleman sayısının karekökü
 - c) Atlama miktarı her zaman 1 olarak kullanılır
 - d) Atlama miktarı her zaman 3 olarak kullanılır
 - e) Atlama miktarı her zaman 5 olarak kullanılır
4. Doğrusal arama algoritmasının en kötü çalışma zamanı aşağıdakilerden hangisidir?
 - a) $O(\log n)$
 - b) $O(n^2)$
 - c) $O(n)$
 - d) $O(1)$
 - e) $O(n^3)$
5. İkili arama algoritmasında 1024 elemana sahip bir dizide en kötü çalışma zamanında yaklaşık kaç karşılaştırma gerekir?
 - a) 1
 - b) 10
 - c) 100
 - d) 500
 - e) 1024
6. Arama işleminde, tahmin için dizinin ilk ve son elemanlarının değerini kullanan arama algoritması aşağıdakilerden hangisidir?
 - a) Hiçbiri
 - b) Doğrusal arama
 - c) Atlamalı arama
 - d) Enterpolasyon arama
 - e) İkili arama

7.

0	64
1	55
2	37
3	21
4	93
5	36
6	15
7	88
8	23
9	41

Yukarıda verilen listede doğrusal arama gerçekleştirildiği varsayılın. 21 değerini bulmak için kaç karşılaştırma işlemi gerekir?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5

8.

0	22
1	35
2	39
3	43
4	47
5	51
6	55
7	59
8	65
9	74
10	87

Yukarıda verilen listede ikili arama gerçekleştirildiği varsayılın. 65 değerini bulmak için kaç karşılaştırma işlemi gerekir?

- a) 2
- b) 3
- c) 4
- d) 5
- e) 6

9. İkili arama algoritmasında hedeflenen çalışma zamanı aşağıdakilerden hangisidir?

- a) $O(1)$
- b) $O(\log n)$
- c) $O(n)$
- d) $O(n \log n)$
- e) $O(n^2)$

10. Doğrusal arama algoritmasında en iyi çalışma zamanı aşağıdakilerden hangisidir?

- a) $O(n^2)$
- b) $O(n \log n)$
- c) $O(n)$
- d) $O(\log n)$
- e) $O(1)$

Yanıt Anahtarı: 1-A 2-C 3-B 4-C 5-B 6-D 7-D 8-A 9-B 10-E

Kaynakça

Thareja R. (2014) Data Structures Using C, 2nd Edition, Oxford University Press.

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

BÖLÜM 9

SIRALAMA ALGORİTMALARI

ANAHTAR KAVRAMLAR

Seçmeli Sıralama, Kabarcık Sıralama, Birleşmeli Sıralama, Hızlı Sıralama

ÖĞRENME HEDEFLERİ

- 1 Kaba kuvvet ve böl-ve-fethet yöntemleri arasındaki farkları anlayabilme
- 2 Çeşitli sıralama algoritmalarının çalışma prensiplerini anlayabilme
- 3 Sıralama algoritmalarının performanslarını karşılaştırabilme

GİRİŞ

Sıralama, verilen elemanları artan veya azalan şeklinde, düzenlemek anlamına gelir. Geçmişten günümüze birçok bilim insanı, sıralama algoritmaları ile ilgilenmişlerdir. Bilgisayar bilimlerinde en çok çalışılan problemlerden biridir. Bazı problem çözümleri, sıralama algoritmalarını, alt bir fonksiyon olarak kullanır. Örneğin n elemanlı bir dizi içerisinde, bir öğenin var olup olmadığını tespit etmek veya n elemanlı bir dizi verildiğinde, öğelerin frekanslarını tespit etmek problemleri, sıralama algoritmaları yardımı ile çözülebilir.

Sıralama algoritmaları en genel haliyle tam sayı ve karşılaştırma temelli yöntemler olmak üzere ikiye ayrılırlar. Karşılaştırma sıralamaları, bir öğenin başka bir öğenin solunda mı yoksa sağında mı olması gerektiğini belirlemek için algoritmanın her adımındaki öğeleri karşılaştırır. Tam sayı ve sayma sıralamaları dizi içerisinde bir öğeden küçük kaç adet öğe olduğu hesap ederek sıralama işlemini gerçekleştirir. Bu yöntemlerin çalışabilmesi için bazı şartların sağlanması gerekmektedir aksi takdirde bu yöntemler uygulanamaz. Bu nedenle bu bölümde herhangi bir şarta bağlı kalmaksızın uygulanabilecekleri için kaba kuvvet ve böl-fethet şeklinde iki başlık altında karşılaştırma temelli sıralama algoritmaları anlatılacaktır.



ARAŞTIRALIM

Bucket ve Radix sıralama algoritmalarının hangi şartlar altında uygulanabildiği ve çalışma prensiplerini öğrenelim.



1. KABA KUVVET SIRALAMA ALGORİTMALARI

Sıralama problemini kaba kuvvet tasarım tekniği kullanarak çözmek için izlenebilecek iki strateji vardır: dizinin küçük öğelerini dizinin başındaki konumlara taşımak, ya da dizinin büyük öğeleri-

ni dizinin sonuna taşımaktır. Bu stratejiler takip edilerek oluşturulmuş algoritmalar sırasıyla Seçmeli sıralama ve Kabarcık sıralaması ismiyle literatürde yer bulmuştur.

1.1. Seçmeli Sıralama

En basit sıralama algoritmalarından biridir. Bu sıralama algoritmasının ilk adımında, dizi içerisindeki en küçük değere sahip eleman bulunur ve dizinin ilk elemanı ile yer değiştirilir. İkinci adımda ise benzer şekilde dizi içerisindeki ikinci en küçük değere sahip eleman bulunur ve dizinin ikinci elemanı ile yer değiştirilir. Bu işlem $(n-1)$ defa uygulanarak dizi elemanlarının tamamının küçükten büyüğe doğru sıralanması sağlanır. Bu

yöntem, kalan en küçük elemanı tekrar tekrar seçerek çalıştığı için seçmeli sıralama olarak adlandırılır. Eğer dizi içerisindeki elemanların büyükten küçüğe doğru bir sıralanması istenirse, her bir adımda kalan en büyük değer bulunması ve yer değiştirilmesi yeterli olacaktır. Seçmeli sıralama algoritmasının sözde kodu Şekil 9.1'de verilmiştir.

```

1.  SelectionSort( $[a_1, a_2, \dots, a_n]$ )
2.  for  $i = 1$  to  $n - 1$ 
3.       $\text{min} \leftarrow i$ 
4.      for  $j = i + 1$  to  $n$ 
5.          if  $a_j < a_{\text{min}}$ 
6.               $\text{min} \leftarrow j$ 
7.          swap  $a_i$  and  $a_{\text{min}}$ 

```

Şekil 9.1 Seçmeli Sıralama Algoritması

Seçmeli sıralama algoritması karışık sıradaki bir diziyi girdi olarak alır ve çıktı olarak küçükten büyüğe sıralı bir şekilde diziyi döndürür. Şekil 9.1'de de görüleceği gibi, algoritma iç içe iki adet for döngüsü ile sıralama işlemini gerçekleştirir. Birinci for döngüsü i değişkenine 1 değerini atar ve $n-1$ değerine ulaşınca kadar çalışır. İkinci for döngüsü j değişkenine $i + 1$ değerini atar ve n değerine ulaşınca çalışmasını tamamlar. İkinci for döngüsü j değişkeninin gösterdiği dizi elemanı ile güncel minimum değerini karşılaştırır. Eğer şu an inceleme altındaki değer küçük ise minimum değeri gösteren indeks güncellenir. Değilse işleme bir sonraki dizi elemanı ile devam edilir. İkinci for döngüsünün tamamlanmasından sonra i indeksinin gösterdiği dizi elemanı ile minimum değer yer değiştirilir.

Tablo 9.1'de sıralı olmayan 8 elemanlı örneklem üzerinde Seçmeli sıralama algoritmasının çalışma adımları gösterilmiştir. Tablonun en sol sütunlarında yer alan i ve min değişken değerleri yöntemin daha kolay anlaşılması için verilmiştir. Kırmızı renkli değerler dizideki en küçük değerleri ve mavi renkli olanlar ise yer değiştirme sonucunda sıralanması tamamlamış değerleri göstermektedir. Yani bu değerler dizi içerisindeki olmaları gereken konumlara yerleştirilmişlerdir. Bundan sonra konumları kati suretle değişmeyecektir.

Tablo 9.1 Seçmeli Sıralama Algoritmasının Çalışma Adımları

i	min	1	2	3	4	5	6	7	8
1	7	8	4	6	9	2	3	1	5
2	5	1	4	6	9	2	3	8	5
3	6	1	2	6	9	4	3	8	5
4	5	1	2	3	9	4	6	8	5
5	8	1	2	3	4	9	6	8	5
6	6	1	2	3	4	5	6	8	9
7	7	1	2	3	4	5	6	8	9
8	8	1	2	3	4	5	6	8	9
		1	2	3	4	5	6	8	9

1.2. Kabarcık Sıralama

Kaba kuvvet tasarım tekniği kullanılarak geliştirilmiş bir diğer sıralama algoritması da Sabarcık sıralama algoritmasıdır. Kabarcık sıralama algoritmasının ilk adımında birinci elemandan başlayarak sonuncu elemana kadar dizinin sırasıyla komşu elemanları karşılaştırılır ve her karşılaştırmada büyük elemanlar dizinin sonuna doğru taşınırlar. Böylece, ilk adımın tamamlanmasıyla en büyük değere sahip eleman dizinin sonuna yerleştirilir. İkinci adımda birinci elemandan

başlayarak sondan bir önceki elemana kadar benzer şekilde dizinin komşuları karşılaştırılır ve yine her karşılaştırmada büyük elemanlar dizinin sonuna doğru taşınırlar. Böylece, ikinci büyük değere sahip eleman dizinin sondan bir önceki konumuna yerleştirilir. Benzer adımları dizide tek bir eleman kalıncaya kadar uygulanır. Kabarcık sıralama algoritmasının sözde kodu Şekil 9.2’de verilmiştir.

```

1. BubbleSort([a1, a2, ..., an])
2. for i = 1 to n - 1
3.   for j = 1 to n - i
4.     if aj+1 < aj
5.       swap aj+1 and aj

```

Şekil 9.2 Kabarcık Sıralama Algoritması

Şekil 9.2’de de görüleceği gibi; Kabarcık sıralama, Seçmeli sıralama algoritmasına benzer bir biçimde iç içe iki adet for döngüsü ile gerçekleştirilmektedir. Dıştaki for döngüsü 1’den n-1’e kadar çalışmaktadır. İçteki for döngüsü 1’den başla-

yıp n - i değeri kadar çalışmaktadır ve if ifadesi ile komşu elemanlar karşılaştırılarak, sırası doğru olmayan elemanların yerlerinin değiştirilmesi sağlanır.

Tablo 9.2 Kabarcık sıralama algoritmasının çalışma adımları

i	j	1	2	3	4	5	6	7	8
		8	4	6	9	2	3	1	5
1	1	4	8	6	9	2	3	1	5
1	2	4	6	8	9	2	3	1	5
1	3	4	6	8	9	2	3	1	5
1	4	4	6	8	2	9	3	1	5

1	5	4	6	8	2	3	9	1	5
1	6	4	6	8	2	3	1	9	5
1	7	4	6	8	2	3	1	5	9
2	1	4	6	8	2	3	1	5	9
2	2	4	6	8	2	3	1	5	9
2	3	4	6	2	8	3	1	5	9
2	4	4	6	2	3	8	1	5	9
2	5	4	6	2	3	1	8	5	9
2	6	4	6	2	3	1	5	8	9
3	1	4	6	2	3	1	5	8	9
3	2	4	2	6	3	1	5	8	9
3	3	4	2	3	6	1	5	8	9
3	4	4	2	3	1	6	5	8	9
3	5	4	2	3	1	5	6	8	9
4	1	2	4	3	1	5	6	8	9
4	2	2	3	4	1	5	6	8	9
4	3	2	3	1	4	5	6	8	9
4	4	2	3	1	4	5	6	8	9
5	1	2	3	1	4	5	6	8	9
5	2	2	1	3	4	5	6	8	9
5	3	2	1	3	4	5	6	8	9
6	1	1	2	3	4	5	6	8	9
6	2	1	2	3	4	5	6	8	9
7	1	1	2	3	4	5	6	8	9
		1	2	3	4	5	6	8	9

Tablo 9.2’de sıralı olmayan 8 elemanlı [8, 4, 6, 9, 2, 3, 1, 5] örneklemini üzerinde Kabarcık sıralama algoritmasının çalışma adımları gösterilmiştir. tam sayı dizisini kabarcık sıralama algoritma ile küçükten büyüğe sıralanması durumundaki bütün adımları gösterilmiştir. Kırmızı değerler karşılaştırılan komşuları ve mavi değerler konumu belirlenen dizi elemanlarını göstermektedir. İçteki for döngüsünün ilk iterasyonunun tamam-

lanmasıyla, dizi içerisindeki en büyük değer olan 9 dizinin en sonuna yerleştirilmiştir. İkinci iterasyonda ikinci en büyük değer 8 dizideki olması gereken konuma taşınmıştır. Sonraki adımlarda dizinin ön kısımlarındaki komşularına göre büyük olan değerlerin dizinin sonlarına doğru taşınmaları sağlanarak dizi küçükten büyüğe sıralı hale getirilmiştir.

2. BÖL - FETHET SIRALAMA ALGORİTMALARI

2.1. Birleşmeli Sıralama

Bir böl-ve-fethet yöntemi olan Birleşmeli sıralama algoritması 3 adımdan oluşur. Birinci adım n boyutlu $X[1...n]$ dizisini $X[1...n/2]$ ve $X[n/2+1...n]$ olmak üzere $n/2$ uzunluğunda iki parçaya böler.

Sonra her bir parça özyinelemeli olarak sıralanır. En son, sıralı olan iki parça birleştirilir. Böylece ana problem çözülmüş olur.

8	4	6	9	2	3	1	5	Sıralı olmayan tam sayı dizisi
4	6	8	9					Sıralı sol parça
				1	2	3	5	Sıralı sağ parça
1	2	3	4	5	6	8	9	Birleştirir metodu ile elde edilen sıralı dizi

Şekil 9.3 Birleşmeli Sıralama Algoritması

Şekil 9.3'te de görüleceği gibi, Birleşmeli sıralama metodu bir dizi ve iki adet tam sayı değerini girdi olarak alır. Burada p parametresi dizinin başlangıç ve r parametresi dizinin bitiş indekslerini göstermektedir. r değeri p den büyük olmak şartıyla q değişkeninin değeri p ile r değerlerinin toplamının ikiye bölünmesi ile elde edilen değer olarak belirlenir. q değeri belirlendikten sonra

Birleşmeli sıralama algoritması dizinin iki parçası üzerinde, yani dizinin p indeksinden q indeksine kadar olan kısmında ve $q + 1$ indeksinden r indeksine kadar olan kısmında iki defa özyinelemeli olarak çağrılır. Son adımda ise birleştirir metodu çağrılarak sıralı halde olan iki dizi parçası birleştirilir.

```

1. Merge( $X[1,n]$ ,  $p$ ,  $q$ ,  $r$ )
2.  $a \leftarrow q - p + 1$ 
3.  $b \leftarrow r - q$ 
4. let  $L[1, a + 1]$  and  $R[1, b + 1]$  be new arrays
5. copy  $X[p, q]$  to  $L[1, a]$ 
6. copy  $X[q+1, r]$  to  $R[1, b]$ 
7.  $L[a+1] \leftarrow \infty$ ;  $R[b+1] \leftarrow \infty$ 
8.  $i \leftarrow 1$ ;  $j \leftarrow 1$ 
9. for  $k = p$  to  $r$ 
10.   if  $L[i] \leq R[j]$ 
11.      $X[k] \leftarrow L[i]$ 
12.      $i \leftarrow i + 1$ 
13.   else
14.      $X[k] \leftarrow R[j]$ 
15.      $j \leftarrow j + 1$ 

```

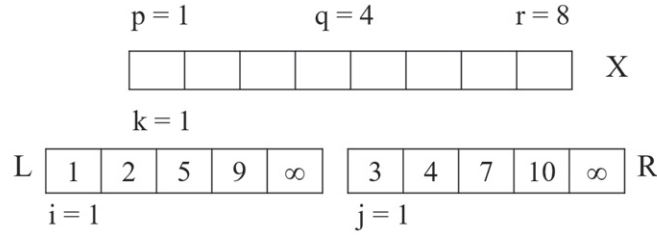
Şekil 9.4 Birleştirir Metodunun Sözde Kodu

**DİKKAT EDELİM**

Özyinelemeli (recursive) fonksiyonların tekrarlamalı(iterative) fonksiyonlara göre farklılıkları vardır. Özyinelemeli fonksiyonların çalışmasını durduracak bir koşulun muhakkak olması gerektiğine dikkat edelim.

Birleşmeli sıralama algoritmasının en kritik metodu birleştirir (Merge). Birleştirir metodu daha küçük boyutlu sıralı iki dizi elemanlarının,

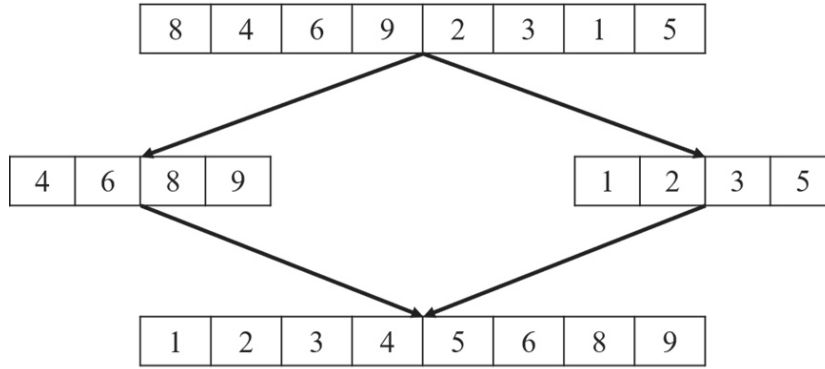
sıralı bir şekilde daha büyük bir diziye yerleştirilmesini gerçekleştirir. Aşağıda Birleştirir metodu örnek bir dizi üzerinde anlatılacaktır.



Şekil 9.5 Birleştirir Metodunun Başlangıç Durumu

Birleştirir metodu a ve b değişkenlerinin değerlerini belirledikten sonra L ve R olmak üzere iki adet yeni dizi oluşturur. X dizisindeki p ve q değişkenlerin arasında kalan değerleri L dizisine ve q+1 ve r aralığındaki değerleri R dizisine kopyalar. Her iki dizinin sonuna oldukça büyük bir değer yerleştirilir. Döngü değişkenleri olan i, j ve k'ye 1 değeri atanır. Böylece i değişkeni L dizisi-

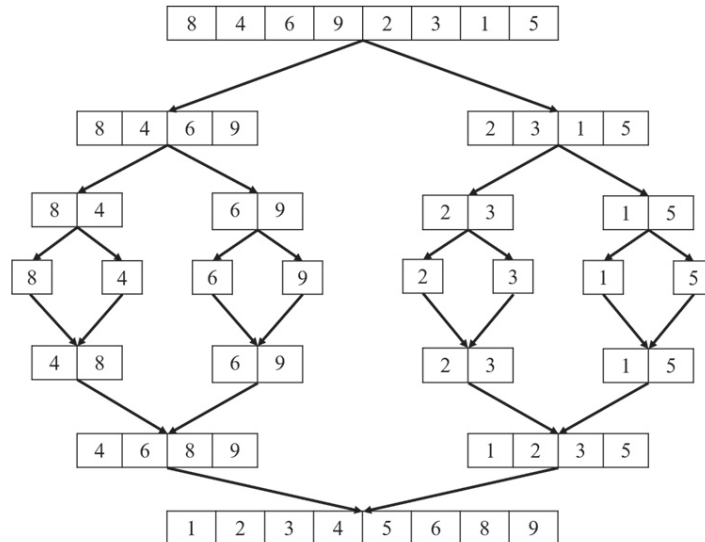
nin, j değişkeni R dizisinin ve k değişkeni X dizisinin başlangıç konumlarını göstermesi sağlanır. Sonraki adımlarda i ve j değişkenlerin gösterdiği değerler karşılaştırılır hangisi küçük ise o değer X dizisini kopyalanır. Her bir kopyalama işleminden sonra k değişkenin değeri bir artırılır. Hangi dizinin elemanı küçük ise onun değişkeni 1 artırılır diğer dizinin değişkeni değiştirilmez.



Şekil 9.6 Birleştirir Metodunun Bir Diziye Uygulanması

Şekil 9.7'de Birleşmeli sıralama algoritmasının [8, 4, 6, 9, 2, 3, 1, 5] örneklemini üzerinde nasıl çalıştığı adım adım gösterilmiştir. İlk olarak dizi elemanları bir boyutlu oluncaya kadar parçalara bölünmüş ve bu bir boyutlu dizi parçaları Birleştirir metodu yardımıyla iki boyutlu sıralı dizi par-

çaları hale getirilmiştir. Daha sonra iki boyutlu parçalar birleştirilmiş ve dört boyutlu sıralı diziler elde edilmiştir. Son olarak ta, bu dört boyutlu parçalar birleştirilmiş ve nihayetinde sıralı bir dizi elde edilmiştir.



Şekil 9.7 Birleşmeli Sıralama Algoritmasının Bir Diziye Uygulanması

2.2. Hızlı Sıralama

Hızlı sıralama algoritmasının çalışma mantığı şöyledir. Dizideki elemanlardan biri pivot olarak belirlenir. Dizinin sol tarafındaki elemanlar belirlenen pivottan küçük ve sağ tarafındaki elemanlar pivottan büyük olacak şekilde dizi iki parçaya ayrılır. Böylece pivot olarak seçilen eleman

olması gereken konuma yerleşirken pivottan küçük değerler sırasız bir şekilde pivottan önce ve büyük değerler pivottan sonraya yerleştirilir. Hem pivottan önceki hem de pivottan sonraki dizi parçaları özyinelemeli olarak sıralanır.

5	4	6	9	2	3	1	8	Sıralı olmayan tam sayı dizisi
3	2	1	4					Pivottan küçükler sol parçaya yerleştirilir
					9	6	8	Pivottan büyükler sağ parçaya yerleştirilir
				5				Pivot olması gereken konuma yerleştirilir
1	2	3	4	5	6	8	9	Sağ ve sol parçalar özyinelemeli olarak sıralanır

Şekil 9.8 Hızlı Sıralama Algoritması

Hızlı sıralama algoritmasının en kritik metodu bölümlenmedir (Partition). Şekil 9.9'da da görüleceği üzere; bu metot, X dizisi, p ve r olmak üzere 3 adet parametre alır. Girdi parametreleri p ve r dizinin başlangıç ve son indekslerini belirlemektedir. Bölümlenme metodunun, çalışmasını tamamlamasıyla, pivot olarak seçilen eleman

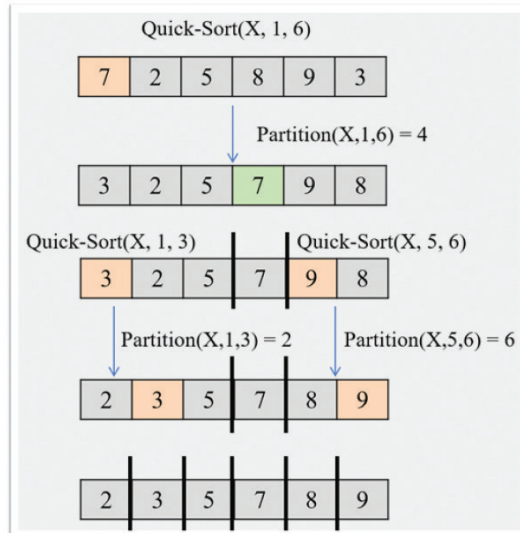
dizi içerisindeki sıralamaya göre, olması gereken yere yerleştirilir. Ayrıca pivottan küçük elemanlar, pivottan önce ve pivottan büyük elemanlar, pivottan sonra olacak şekilde dizide yerleştirilir. Bunlara ek olarak pivot elemanın dizideki konumu metot tarafından döndürülür.

```

1. Partition(X,p,r)
2. k ← ap ; s ← p
3. for i = p + 1 to r
4.   if ai < k
5.     s ← s + 1 ; swap(as, ai)
6. swap(ap, as)
7. return s

```

Şekil 9.9 Bölümlenme Metodunun Sözde Kodu



Şekil 9.10 Hızlı Sıralama Algoritmasının Örnek bir dizi üzerinde çalışma adımları

Şekil 9.10'da Hızla sıralama algoritmasının ve-rilen bir girdi üzerinde nasıl çalıştığı adım adım sıralı diziyi nasıl oluşturduğu gösterilmiştir. Şekilde de görüleceği üzere, ilk bölümleme metodu uygulandıktan sonra pivot dizide dördüncü konuma taşınmış ve kendisinden küçüklerin sırasız bir şekilde bölümlemenin sol tarafında ve büyüklerin ise sağ tarafında olması sağlanmıştır. Hızlı sıralama metodu bir sonraki adımda özyinemeli olarak (1...3) ve (5...6) dizi parçaları için çağrılmıştır. Sol parça için pivot değeri olarak 3 ve sağ parça için pivot değeri olarak 9 alınmıştır. En son aşamada her bir parça bir elemanlı dizi parçalarından meydana geldiği için Hızlı sıralama metodu sonlandırılmıştır.

Algoritmanın çalışma süresi, bölümlemenin dengeli veya dengesiz olmasına bağlıdır. Bölüm-

leme dengeli ise, algoritma birleştirme sıralaması kadar hızlı çalışır ama dengesiz ise daha yavaş çalışabilir. En kötü durum dizinin en başta sıralı (küçükten büyüğe veya büyükten küçüğe olması fark etmez) bir şekilde olması durumunda ortaya çıkar. Bölümleme metodu her adımda sıfır elemanlı bir alt problem ve bir eleman daha az elemanlı başka bir alt problem oluşturur. Bu durumda algoritmanın çalışma süresi $O(n^2)$ olarak ortaya çıkar. Bu süre kabarcık ve seçmeli sıralama algoritmalarının çalışma sürelerine eşittir. En iyi durum yöntemin her seferinde $n / 2$ elemanlı alt problemler oluşturmasında gözlemlenir. Bu durumda hızlı sıralama algoritması, birleşmeli sıralama ile aynı zamanda sıralama işlemini gerçekleştirir.

3. SIRALAMA ALGORİTMALARININ KARŞILAŞTIRILMASI

Hızlı ve Kabarcık sıralama algoritmaları gibi bazı sıralama algoritmaları girdi değerlerine bağlı olarak farklı performans gösterebilmektedirler. Birleşmeli sıralama algoritmaları gibi bazıları ise girdi değerlerinden bağımsız olarak her durumda aynı performansı sergileyebilmektedirler. Sıralama algoritmalarının performanslarını

belirlerken gerçekleştirdikleri hem toplam karşılaştırma hem de yer değiştirme sayıları dikkate alınır. Yer değiştirme maliyeti çoğu durumda karşılaştırma maliyetinin yanında ihmal edilebilir seviyede olduğu için Tablo 9.3'de sadece sıralama algoritmalarının karşılaştırma maliyetleri açısından değerlendirmeleri gerçekleştirilmiştir.

Tablo 9.3 Sıralama Algoritmalarının Performans Karşılaştırılması

Sıralama Algoritması	Ortalama durum (Average)	En iyi durum (Best)	En kötü durum (Worst)
Seçmeli sıralama	$O(n^2)$	$O(n^2)$	$O(n^2)$
Kabarcık sıralama	$O(n^2)$	$O(n^2)$	$O(n^2)$
Birleşmeli sıralama	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n \cdot \log n)$
Hızlı sıralama	$O(n \cdot \log n)$	$O(n \cdot \log n)$	$O(n^2)$

Seçmeli ve Kabarcık sıralama algoritmaları girdi elemanlarının durumlarına bakmaksızın her durumda $O(n^2)$ kadar karşılaştırma yapmaları gerekmektedir ama yer değiştirme açısından durum farklıdır. Eğer dizi büyükten küçüğe sıralı ise bu durumda Kabarcık sıralaması en fazla yer değiştirme işlemi gerçekleştirmek zorundadır. Seçmeli sıralama benzer durumda daha az yer

değiştirme yapması yeterli olacaktır. Hızlı sıralama yukarıda da bahsedildiği üzere pivot elemanın seçimine bağlı olarak dengeli veya dengesiz parçalar oluşturabilmektedir. Bu durum aynı boyutlu diziler için farklı çalışma sürelerine sebep olabilmektedir. Fakat Birleşmeli sıralama için böyle bir durum söz konusu değildir.

BÖLÜM ÖZETİ

Sıralama veri işlemede ve modern bilimsel hesaplamada önemli bir rol oynar. Aynı zamanda sonraki bölümlerde anlatılacak olan Kruskal algoritması gibi birçok problemin çözümünde ara işlem olarak uygulanır. Bu bölümde Seçmeli, Kabarcık, Birleşmeli ve Hızlı olmak üzere dört adet karşılaştırma temelli sıralama algoritması açıklanmıştır. Seçmeli ve Kabarcık sıralama algoritmaları kaba kuvvet tekniğini kullanarak sıralama işlemini gerçekleştirmektedir. Kaba kuvvet yöntemleri bir problemi çözmek için problemin çözüm kümesinin bütün olası durumlarını dikkate aldığı için çoğu durumda diğer tekniklere göre yavaş çalışma sürelerine sahiptir. Böl ve fethet yöntemleri bir problemi genel olarak üç adımda çözerler. Birinci adımda problem yaklaşık eşit büyüklükte küçük parçalara ayrılır. Her bir küçük parça uygun bir şekilde çözülür ve son adımda çözülmüş olan parçalar birleştirilir. Böylece büyük parçanın çözümü elde edilmiş olur. Birleşmeli ve Hızlı sıralama algoritmaları böl-ve-fethet tekniği ile çalışmaktadırlar. Sıralama algoritmaları yaptıkları karşılaştırma ve yer değiştirme sayıları, kullandıkları ekstra bellek alanı miktarı ve kararlı olup olmadıkları açısından karşılaştırılırlar. Kabarcık ve Seçmeli sıralama en basit sıralama algoritmaları arasındadır ama en yüksek zaman karmaşıklığı değerlerine sahiptirler. Ancak, her iki algoritmanın da bellek kullanımları düşüktür. Birleşmeli ve Hızlı sıralama algoritmaları daha az zaman karmaşıklığına sahiptirler. Hızlı sıralama, diğer Birleşmeli sıralama algoritmasına göre daha az miktarda bellek kullandığı için daha verimli bir sıralama algoritmasıdır.



ARAŞTIRALIM

Bu bölümde 4 adet temel sıralama algoritması açıklanmıştır ama daha birçok sıralama algoritması mevcuttur. Burada açıklanmayan algoritmaları farklı kaynaklardan araştırılabilir.

GÖZDEN GEÇİRELİM

1. Hızlı sıralama algoritması kullanarak sekiz tam sayıdan oluşan bir diziyi sıraladığımızı varsayalım. İlk bölümlene metodunun gerçekleştirilmesinden sonra dizinin görünümü aşağıda gibi olduğuna göre hangi değer pivot olarak kullanılmıştır?

5	8	4	7	15	13	12	9
---	---	---	---	----	----	----	---

- a) 7
b) 8
c) 12
d) 13
e) 15

2. Aşağıda verilen sıralama algoritmalarından hangilerinin ortalama durum (average-case) zaman karmaşıklığı $O(n^2)$ 'dir?

- I. Seçmeli (Selection) sıralama
II. Hızlı (Quick) sıralama
III. Birleşmeli (Merge) sıralama

- a) Yalnız I
b) Yalnız II
c) I ve II
d) I ve III
e) I, II ve III

3. Aşağıda verilen sıralama algoritmalarından hangilerinin en iyi durum (best-case) zaman karmaşıklığı $O(n^2)$ 'dir?

- I. Seçmeli (Selection) sıralama
II. Hızlı (Quick) sıralama
III. Birleşmeli (Merge) sıralama

- a) Yalnız I
b) Yalnız II
c) I ve II
d) II ve III
e) I, II ve III

4. Aşağıda verilen sıralama algoritmalarından hangilerinin en kötü durum (worst-case) zaman karmaşıklığına $O(n \cdot \log n)$ 'dir?

- I. Seçmeli (Selection) sıralama
II. Birleşmeli (Merge) sıralama
III. Hızlı (Quick) sıralama

- a) Yalnız I
b) Yalnız II
c) I ve II
d) I ve III
e) I, II ve III

5. Aşağıda verilen sıralama algoritmalarından hangileri böl-ve-fethet tasarım tekniği kullanarak geliştirilmiştir?

- I. Seçmeli (Selection) sıralama
II. Kabarcık (Bubble) sıralama
III. Birleşmeli (Merge) sıralama

- a) Yalnız I
b) Yalnız II
c) Yalnız III
d) I ve III
e) II ve III

6. Aşağıdaki dizi durumlarından hangisi küçükten büyüğe artan sıralama probleminde Kabarcık sıralama algoritması kullanıldığında, ilk döngü dizide elemanların tamamı üzerinde işlemini tamamladığında elde edilebilir?

- a) [7, 9, 3, 5, 1]
b) [5, 7, 9, 3, 1]
c) [3, 5, 9, 1, 7]
d) [3, 7, 1, 5, 9]
e) [9, 5, 7, 1, 3]

7. Aşağıda verilen tam sayı dizileri Kabarcık sıralama algoritması kullanılarak küçükten büyüğe doğru sıralanacaktır. Bu durumda en fazla yer değiştirme işlemi hangi seçenekte verilen dizi için gerçekleştirilir?

- a) [9, 7, 3, 1, 5]
- b) [5, 7, 9, 3, 1]
- c) [3, 5, 9, 1, 7]
- d) [3, 7, 1, 5, 9]
- e) [9, 7, 5, 3, 1]

9. Aşağıda verilen sıralama algoritmalarından hangileri kaba kuvvet tasarım tekniği kullanarak geliştirilmiştir?

- I. Seçmeli (Selection) sıralama
- II. Kabarcık (Bubble) sıralama
- III. Birleşmeli (Merge) sıralama

- a) Yalnız I
- b) Yalnız II
- c) I ve II
- d) II ve III
- e) I, II ve III

8. Aşağıdakilerden hangisi küçükten büyüğe artan sıralama probleminde Seçmeli sıralama algoritması kullanıldığında, ilk döngü dizide elemanların tamamı üzerinde işlemini tamamladığında elde edilebilir?

- a) [3, 5, 1, 9, 7]
- b) [3, 7, 1, 9, 5]
- c) [5, 3, 7, 9, 1]
- d) [1, 5, 9, 3, 7]
- e) [7, 3, 1, 9, 5]

10. Aşağıdakilerden hangisi bir karşılaştırma temelli sıralama algoritması değildir?

- a) Seçmeli (Selection) sıralama
- b) Kabarcık (Bubble) sıralama
- c) Birleşmeli (Merge) sıralama
- d) Hızlı (Quick) sıralama
- e) Kök (Radix) sıralama

Yanıt Anahtarı: 1-E 2-A 3-A 4-B 5-C 6-D 7-E 8-D 9-C 10- E

Kaynakça

- Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.
- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.
- ÇÖLKESİN T.R. (2019) Veri Yapıları ve Algoritmalar, Papatya Yayıncılık
- Sedgewick R., Wayne K. (2014) Algorithms, 4th Edition, Addison-Wesley
- Çamoğlu K. (2021) Algoritma, Kodlab

BÖLÜM 10

ÇİZGELER

ANAHTAR KAVRAMLAR

Komşuluk Listesi, Komşuluk Matrisi,
Önce Genişlik Araması, Önce Derinlik Araması

ÖĞRENME HEDEFLERİ

- 1 Çizge teorisine ait temel kavramları tanımlayabilir
- 2 Çizge veri yapısını bilgisayar ortamında temsil edebilir
- 3 Çizge üzerinde dolaşma yöntemlerini uygulayabilir

GİRİŞ

Çizge teorisinin ilk örneği 1736'da "Königsberg'in Yedi Köprüsü" problemini çözen Euler tarafından sunulmuştur. Euler, şehri ve köprülerini temsil eden bir çizge veri yapısı oluşturarak ve bir köprüyü iki kez geçmeden bütün şehri dolaşmanın imkânsız olduğunu kanıtlamıştır. Bu çalışma, çizge teorisinin gelişimi ve bilgisayar bilimindeki ilk uygulamalarının temelini oluşturmuştur.

Günümüzde çizge teorisi bilgisayar bilimi ve mühendisliğin önemli bir parçasıdır ve çizge teorisinin uygulamaları sosyal ağlar, doğal dil işleme, biyoinformatik ve ağ güvenliği gibi birçok alanda bulunabilir. Çizgeler ayrıca bilgisayar biliminde ağaçlar ve bağlantılı listeler gibi veri yapılarını temsil etmek için de kullanılır. Çizge teorisi algoritmaları, ağların yapısını analiz etmek, düğümler arasındaki en kısa yolları belirlemek ve veri aktarımı için en verimli yolları bulmak için kullanılır. Ayrıca sosyal ağlardaki kullanıcıların birbirleriyle olan ilişkileri bir çizge veri yapısı kullanarak temsil edilmekte ve ortaya çıkan yapının özellikleri araştırılmaktadır.

Çizge dolaşma yöntemleri, çizgeye ait her bir düğümü bir düzene bağlı kalarak ziyaret etmeyi amaçlamaktadır. Çizge dolaşma algoritmaları, düğümler arasındaki yolları bulmak, döngüleri tespit etmek ve iki düğüm arasındaki en kısa yolu belirlemek için kullanılır. Önce genişlik ve önce derinlik olmak üzere iki farklı çizge dolaşma yöntemi tanımlanmıştır. Her iki yöntemde farklı stratejiler kullanarak çizgede bütün düğümleri ve kenarları keşfetmektedirler.



ARAŞTIRILIM

Çizge veri yapılarının farklı alanlarda kullanım yerlerini araştırılabilir.

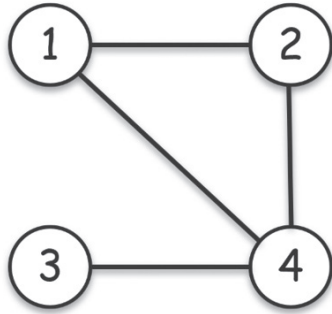


1. ÇİZGE TEORİSİ

Çizge nesneler arasındaki karmaşık ilişkileri temsil etmek ve analiz etmek için kullanılan önemli bir araçtır. Çizge düğümler ve her biri bir çift düğümü birbirine bağlayan kenarlar kümesinden oluşur. Bir çizge $G = (V, E)$ iki farklı yapı kullanılarak şekilde temsil edilebilmektedir: komşuluk listesi veya komşuluk matrisi. Komşuluk listesi çizgeye ait kenar sayısının az olması durumunda tercih edilir.

Çizgeler, düğümler arasındaki ilişkiye bağlı olarak kenarlar için yön belirlenip belirlenme durumuna göre ikiye ayrılmaktadır: yönsüz

çizge ve yönlü çizge. Örneğin, bir ülkedeki farklı şehirler arasındaki bağlantıları gösteren bir çizge, yönsüz bir çizge olacaktır. Bu grafikte, iki şehir arasındaki bağlantı, bu iki şehir arasında bir yol veya başka bir ulaşım olduğunu gösterir, ancak bağlantının yönü önemli olmaz. Yönlü çizgede kenarların yönünün önemi vardır. Bir kenarın hangi düğümden başlayıp hangi düğüme ulaştığı kenarın yönünü belirler. Şehir içindeki bir yolun sadece tek bir yönden trafiğe izin vermesi yönlü bir kenar ile temsil edilir. Bu durumda sadece başlangıç düğümünden diğer düğüm çiftine ulaşmak söz konusudur.



a) Yönsüz bir çizge

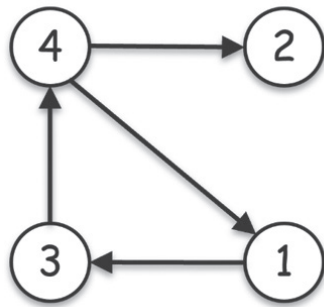
1 - 2,4
2 - 1,4
3 - 4
4 - 1,2,3

b) Komşuluk listesi

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

c) Komşuluk matrisi

Şekil 10.1 Yönsüz çizge örneği ve komşuluk listesi ve matrisi gösterimi



a) Yönlü bir çizge

1 - 3
2 -
3 - 4
4 - 1,2

b) Komşuluk listesi

	1	2	3	4
1	0	0	1	0
2	0	0	0	0
3	0	0	0	1
4	1	1	0	0

c) Komşuluk matrisi

Şekil 10.2 Yönlü çizge örneği ve komşuluk listesi ve matrisi gösterimi



DİKKAT EDELİM

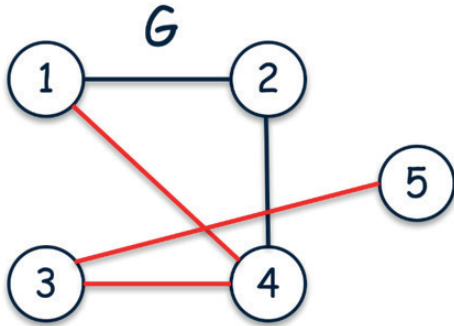
Bir yönlü çizgenin komşuluk matrisi simetrik iken, yönsüz çizgenin komşuluk matrisi simetrik olmayabilir.

Komşuluk listesinde; her bir düğüm için, içinde ilgili düğümden gidilebilecek diğer düğümlerin, yani ilgili düğümün komşularının tutulduğu bir liste oluşturulur. Komşuluk matrisinde ise; n dizinin düğüm sayısı olmak üzere $n \times n$ boyutlu bir matris oluşturulur ve eğer i . ve j . düğümler çizgede komşuysa matrisin (i, j) elemanı 1 değerini, değilse 0 değerini alır. Hem satırda hem de sütunda düğümleri göstermektedir.

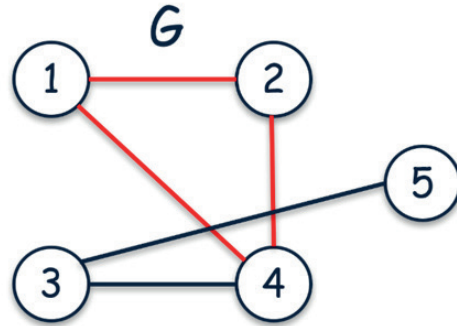
Çizgelerde düğümler arasındaki ilişkiler farklıysa ve bu çizgeye yansıtılmak istemiyorsa, kenarlara bu ilişkinin yoğunluğuna göre bir ağırlık değeri atanabilmektedir. Böyle oluşturulan çizgeler ağırlıklı çizge olarak tanımlanmaktadır. Diğer yandan çizgelerde düğümlerin derecesi düğümlerle bağlantılı kenarlar üzerinden belirlenen komşu sayısı olarak ifade edilmektedir. Yönlü çizgede, düğümlerle bağlantılı kenarlar yönlü olduğundan iki farklı derece tanımlanmaktadır: gir-

di derecesi (düğümü bitiş düğümü olarak kabul eden kenarların sayısı) ve çıktı derecesi (düğümü başlangıç düğümü olarak kabul eden kenarların sayısı).

Bir çizgede yol, dizideki ardışık her iki düğüm için çizgede bir kenar olması koşuluyla sıralı bir düğümler dizisi olarak tanımlanmaktadır. Herhangi bir yolun uzunluğu ise yoldaki toplam kenar sayısıdır. Dizide bütün düğümlerin farklı olması durumunda yol basit yol olarak tanımlanır. İki farklı düğüm arasında bir yol var ise düğümler bağlı (connected) olarak adlandırılırlar. Bir çizgede bütün düğüm çiftleri arasında bir yol olması durumunda çizgeye bağlı çizge denir. Bir grafikteki döngü, aynı düğümden başlayan ve biten bir yoldur. Şekil 10.3.a'da 1, 4, 3, 5 düğümlerinden oluşan yol basit bir yoldur ama Şekil 10.3.b'de 1, 2, 4, 1 yolu bir döngüdür.



a) Basit yol örneği

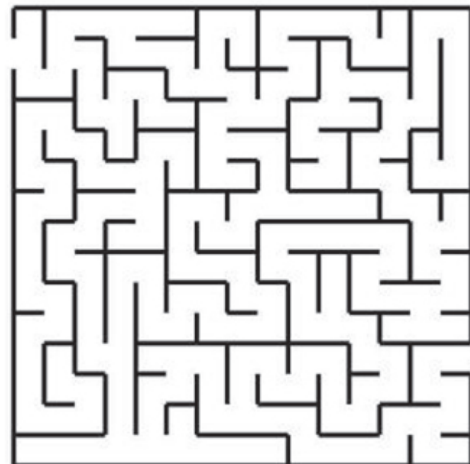


b) Döngü örneği

Şekil 10.3 Basit yol ve döngü örnekleri

2. ÇİZGE DOLAŞMA YÖNTEMLERİ

En temel çizge problemlerinden biri çizge üzerindeki her kenarı ve düğümü dolaşmaktır. Buradaki amaç çizgeyi sistematik bir şekilde keşfetmektir. Etkili bir çizge dolaşma metodu Şekil 10.4'te verilen labirent üzerinden daha iyi anlaşılacaktır. Labirentteki her bir kavşağı bir düğümler ve her koridoru da bir kenarla temsil edelim ve çizge dolaşma yöntemi kullanarak çıkışı bulmaya çalışalım. Burada elbette çok zaman kaybedilmeden çıkışı bulurken aynı yerler tekrar tekrar ziyaret edilmemelidir. Yani çizge dolaşma problemlerinde, aynı düğüm ve kenarlar tekrar tekrar ziyaret edilmemelidir.



Şekil 10.4 Labirent örneği

Bir çizge dolaşma yönteminin iki adet anahtar işlemi bulunmaktadır. Bunlardan birincisi ziyaret ettiğinizde her düğümü işaretlemek ve ikincisi ise henüz tam olarak keşfetmediğiniz düğümleri takip etmek. Algoritmanın çalıştırılması esnasında, her düğüm için olası 3 durum bulunmaktadır. Birinci durum *Keşfedilmemiş*, ikinci durum *Keşfedilmiş* ve üçüncü durum *İşlenmiş* olarak

ifade edilebilir. Keşfedilmemiş durum bir düğüm için başlangıç durumudur. Keşfedilmiş durumda algoritmanın düğümü ziyaret ettiği ama düğümle işinin henüz bitmediği anlaşılmaktadır. Son olarak İşlenmiş durumda ise, algoritmanın düğümü ziyaret ettiği ve bir daha düğümü ziyaret etmeyecek şekilde işlediği anlaşılmaktadır.

2.1. Önce Genişlik Arama (Breadth First Search)

Önce genişlik arama yönteminin amacı öncelikli olarak bir düğümü ve düğümün bütün komşularını keşfetmektir. Bu yöntemde, aynı seviyedeki düğümlerin tamamı işlendikten sonra ancak bir sonraki seviyedeki düğümlere geçilecektir. Bu yöntemde çizge, atomda çekirdek etrafında elektronların oluşturduğu yörüngeler gibi, merkez kabul edilen bir düğüm etrafındaki farklı katmanlara yerleştirilmiş düğümler olarak ele alınmakta ve düğümler en içteki yörüngeden en dıştaki yörüngeye doğru enlemesine bir şekilde keşfedilmektedir.

Yöntemde, çizge üzerindeki her bir u düğümü mesafe, ebeveyn ve renk olmak üzere üç parametre ile tanımlanır. Mesafe merkez s düğümünden u düğüme olan en kısa yolun uzunluğudur. Ebeveyn ise s 'den u 'ya giden en kısa yolda u 'nun selefi olan düğümü ifade eder. Renk parametresi de u düğümünün yöntem esnasında durumunu gösterir. Beyaz renk düğümün keşfedilmemiş, gri renk keşfedilmiş ve siyah renk ise düğümün işlenmiş olduğunu ifade eder.

```

1. BFS(G,s)
2. for each vertex  $u$  of  $V$ 
3.      $u.color = white$ 
4.      $u.dis = \infty$ 
5.      $u.par = nil$ 
6.  $s.color = gray$ 
7.  $s.dis = 0$ 
8. initialize an empty queue  $Q$ 
9. Enqueue( $Q,s$ )
10. while  $Q \neq \emptyset$ 
11.      $u = Dequeue(Q)$ 
12.     for each  $v \in Adj(u)$ 
13.         if  $v.color = white$ 
14.              $v.color = gray$ 
15.              $v.dis = u.dis + 1$ 
16.              $v.par = u$ 
17.             Enqueue( $Q,v$ )
18.      $u.color = black$ 

```

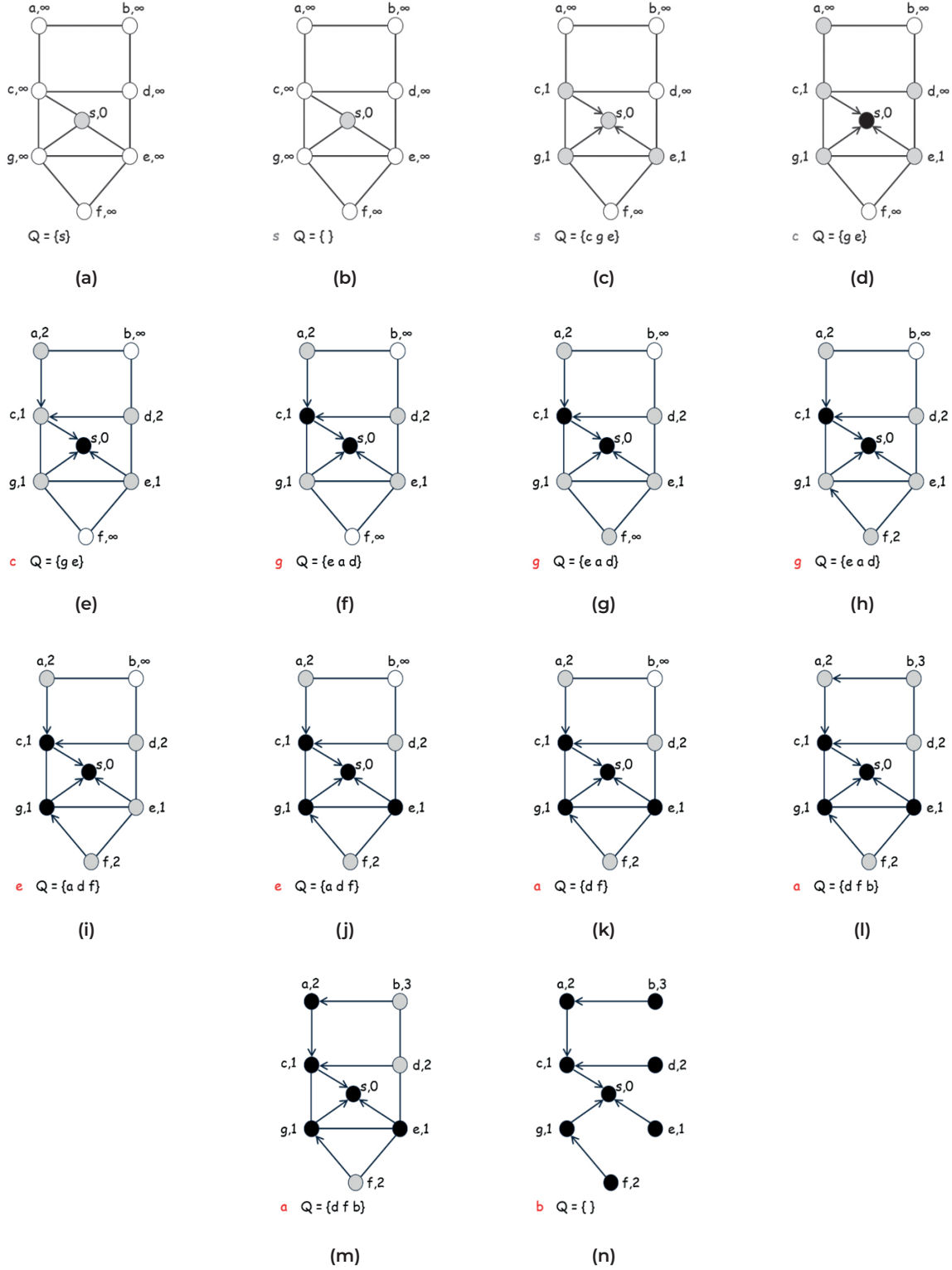
Şekil 10.5 Önce Genişlik Arama yöntemine ait sözde kod

Şekil 10.5'te önce genişlik aramasına ait sözde kod verilmiştir. Kod incelendiğinde ilk olarak bütün düğümlerin renk, mesafe ve ebeveyn parametrelerine sırasıyla beyaz, sonsuz ve null başlangıç değerleri atanır. Sonrasında merkez s düğümünün rengi gri ve mesafesi 0 olarak atanır. Daha sonrasında bir boş kuyruk yapısı oluş-

turarak başlangıç düğümü kuyruğa eklenmiştir. Oluşturulan döngü yapısı kuyruk dolu olduğu sürece aşağıdaki işlemleri gerçekleştirecektir: (i) ilk olarak kuyruğun başındaki düğümü kuyruktan çıkartarak u değişkenine atayacak, (ii) bu düğümün bütün beyaz komşularının rengini gri yapacak, mesafesini bir artıracak ve ebeveynini

u düğümü olarak belirleyecek, (iii) ilgili komşu düğümleri kuyruğun sonuna ekleyecektir. Kuyruktaki bütün düğümler gezilerek işlenecektir.

Kuyruktaki bütün düğümlerin gezilmesi ile önce genişlik arama yöntemi tamamlanacaktır.



Şekil 10.6 Önce Genişlik Arama algoritmasının örnek bir çizge üzerinde uygulanması

Önce genişlik arama algoritması örnek bir çizge kullanılarak bütün adımları Şekil 10.6'da gösterilmiştir. Yöntem esnasında:

(a) Başlangıç düğümü olarak s düğümü seçilmiş, rengi gri, ebeveyni null ve mesafe değeri 0 olarak atanmış ve kuyruğa eklenmiştir.

(b) Kuyruğun başındaki tek düğüm olan s çıkartılmıştır.

(c) S düğümünün beyaz komşuları olan c, g ve e düğümleri kuyruğa eklenmiş, mesafe değerleri 1, renkleri gri ve ebeveynleri s düğümü olarak atanmıştır.

(d) C düğümü kuyruktan çıkartılmıştır.

(e) C düğümünün beyaz komşuları olan a ve d düğümleri keşfedilmiş ve renkleri gri ve mesafe değerleri 2 olarak atanmıştır.

(f) C düğümü siyaha boyanmıştır ve g düğümü kuyruktan çıkartılmıştır.

(g) G düğümünün beyaz komşusu f düğümü keşfedilmiş ve rengi gri yapılmıştır.

(h) F düğümünün mesafe değeri 2 olarak belirlenmiştir.

(i) G düğümü siyaha boyanmıştır ve e düğümü kuyruktan çıkarılmıştır.

(j) E düğümünün hiçbir komşusu beyaz olmadığı için siyaha boyanmıştır.

(k) Kuyruğun başındaki a düğümü çıkartılmıştır.

(l) B düğümü kuyruğa eklenerek mesafe değeri olarak 3 atanmıştır.

(m) A düğümü siyaha boyanmıştır. Gri renkli olan d, f ve b düğümleri sıra ile kuyruktan çıkarılmış ve çizgede başka beyaz düğüm kalmadığı için siyaha boyanmışlardır.

(n) Düğümlerin mesafe değerleri ve ebeveynlerini gösteren öncelikli arama ağacı elde edilmiştir.

2.2. Önce Derinlik Arama (Depth First Search)

Önce derinlik arama yöntemi, önce genişlik arama algoritmasından farklı olarak çizgenin düğümlerini enlemesine değil derinlemesine keşfeder. Yöntemde bir u düğümünü keşfedildiğinde, u'nun keşfedilmemiş bir v komşusunu seçilerek aramaya v ile devam edilir. Sonrasında u'nun diğer komşularını ziyaret etmek yerine v'nin ziyaret edilmemiş bir komşusu seçilerek aramaya ilgili komşu ile devam edilir. Bu şekilde ancak v komşusundan ulaşılacak tüm düğümler ziyaret edildikten sonra v işlenmiş kabul edilir ve u'nun bir diğer ziyaret edilmemiş komşusuna geçilir. İşlem, başlangıç düğümünden ulaşılabilen tüm düğümler keşfedilene kadar devam eder. Çizgede keşfedilmemiş düğümler kalırsa, bunlardan biri yeni başlangıç düğümü olarak seçilir ve benzer adımları tekrarlar.

Çizge üzerindeki her bir u düğümünün keşif ve bitiş zamanı, ebeveyn ve renk olmak üzere dört parametresi vardır. Ebeveyn ve renk parametreleri önce genişlik arama yöntemindeki gibidir. Önce genişlik arama yönteminden farklı olarak burada bir zaman sayacı vardır ve her keşfetme veya işleme işleminde bu sayaç artırılır. Düğümlerin keşif zamanı düğümü ilk keşfettiğimiz zamanı ve bitiş zamanı ise düğümün işlenme zamanını gösterir.

Önce derinlik arama yöntemini iki bölüme ayırmak mümkündür. Birinci bölüm her bir düğümün parametrelerin atandığı ve çizgede beyaz düğüm olduğu sürece çağrılacak olan “Visit” fonksiyonundan oluşmaktadır.

```

1. DFS_Visit(u)
2. u.color = gray
3. time = time + 1
4. u.dis = time
5. for each v in Adj(u)
6.     if (v.color = white)
7.         v.par = u
8.         DFS_Visit(v)
9. u.color = black
10. time = time + 1
11. u.fin = time

```

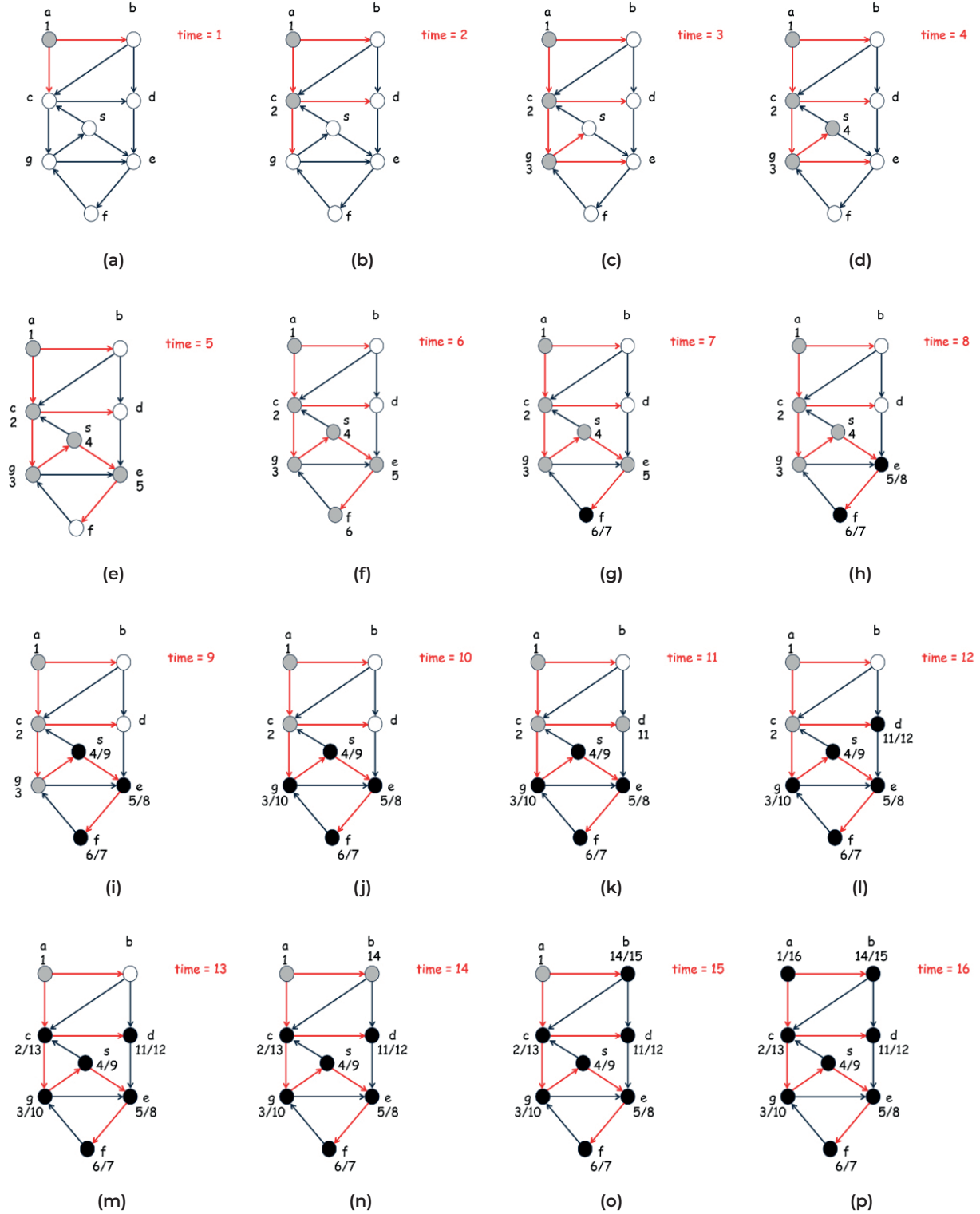
Şekil 10.7 Visit yöntemine ait sözde kod

İkinci bölüm ziyaret edilen düğümün rengini öncelikli olarak gri yapar ve zaman değişkenini bir artırır ve düğümün keşif zamanı olarak

atar. Sonrasında her bir beyaz komşu için ziyaret fonksiyonu tekrar çağrılır. Keşfedilen düğümden

gidilecek beyaz düğüm kalmaz ise rengini siyaha boyar, zamanı bir artırır ve bitiş değerine atar.

Şekil 10.8 Önce derilik arama yöntemine ait sözde kod



Şekil 10.9 Önce Derinlik arama algoritmasının örnek bir çizge üzerinde uygulanması

Önce derinlik arama algoritması örnek bir çizge kullanılarak bütün adımları Şekil 10.9'da gösterilmiştir. Yöntem esnasında:

- (a) Başlangıç düğümü olarak a belirlenmiş, keşif zamanı 1 ve rengi gri yapılmıştır.
- (b) Beyaz komşulardan biri olan c düğümü seçilerek rengi gri ve keşif zamanı 2 olarak belirlenmiştir.
- (c) A düğümü keşfedilerek rengi gri ve keşif zamanı 3 olarak belirlenmiştir.
- (d) S düğümü keşfedilmiş, rengi gri ve keşif zamanı olarak 4 atanmıştır.
- (e) S düğümünden gidilebilecek ve beyaz komşusu olan e düğümü ziyaret edilmiştir.
- (f) F düğümü ziyaret edilerek rengi griye boyanmış ve keşif zamanı olarak 6 atanmıştır.
- (g) F düğümünde gidilebilecek başka bir düğüm olmadığı için rengi siyaha boyanarak bitiş zaman değeri 7 atanmıştır.
- (h) E düğümüne geri dönülerek ve ziyaret edilebilecek başka bir düğüm olmadığı için siyaha boyanmıştır. Bitiş zamanı olarak 8 değeri atanmıştır.

(i) S düğümüne geri dönmüş ve rengi siyaha boyanarak bitiş zamanı 9 olarak belirlenmiştir.

(j) C düğümüne geri dönmüş ve d düğümü ziyaret edilmiştir.

(k) D düğümün keşif zamanı olarak 11 belirlenmiştir.

(l) D düğümünden gidilebilecek başka bir düğüm olmadığı için bitiş zamanı 12 olarak atanmıştır.

(m) C düğümüne geri dönmüş ve başka ziyaret edilebilecek düğüm kalmadığı için bitiş zamanı 13 olarak atanmıştır.

(n) A düğümüne geri dönmüş ve A'nın bir diğer komşusu B düğümü keşfedilerek rengi griye boyanmış ve keşif zamanı 14 belirlenmiştir.

(o) B düğümünde gidilebilecek başka bir düğüm olmadığı için siyaha boyanmış ve bitiş zaman değeri 15 atanmıştır.

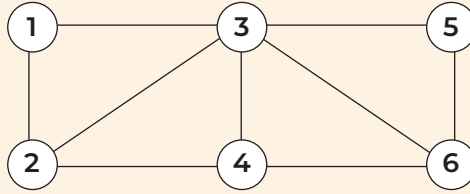
(p) A düğümüne geri dönmüş ve başka bir düğüm kalmadığı için siyaha boyanmış ve bitiş değeri 16 atanmıştır.

BÖLÜM ÖZETİ

Çizgeler çeşitli nesneler arasındaki ilişkileri temsil etmek için kullanılan bir veri yapısıdır. Bir çizge, düğümler arasındaki bağlantıları temsil eden bir düğümler ve kenarlar kümesidir. Çizgeler, sosyal ağlar, matematiksel denklemler ve dil yapıları dahil olmak üzere çok çeşitli veri türlerini temsil etmek için kullanılabilir. Bu bölümde ilk olarak çizge teorisine ait temel kavramlar açıklanmıştır. Çizgeye ait kenarların yönlerinin önemli olması durumunda yönlü çizge ve önemli olmaması durumunda yönsüz çizge türleri açıklanmıştır. Ayrıca bir çizge veri yapısını bilgisayar ortamında temsil edilebilmesi için gerekli komşuluk listesi ve matrisi anlatılmıştır. Çizge dolaşma algoritmaları, bir düğümden başlayarak ve çizgedeki tüm düğümleri belirli bir sırada ziyaret etmek için kullanılır. Bu algoritmalar, çizge veri yapısının aranmasına ve sıralanmasına izin verir. Yaygın çizge dolaşma algoritmaları, önce derinlik ve önce genişlik aramadır. Bu algoritmalar, çizge üzerinde dolaşmak için farklı stratejiler kullanır ve farklı çizge türlerine uygulanabilir. Önce derinlik arama algoritması bir kuyruk veri yapısı yardımıyla düğümlerin hangi sırayla dolaşılması gerektiğine karar verir. Buna karşın önce derinlik arama algoritması bir yığın veri yapısıyla gezinme işlemini gerçekleştirir.

GÖZDEN GEÇİRELİM

1. Aşağıda verilen çizgeye ait komşuluk matrisi gösterimi hangi seçenekte doğru verilmiştir?



a)

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	1	0	1	1	0	0
3	1	1	0	1	1	1
4	0	1	1	0	0	1
5	0	0	1	0	0	1
6	0	0	1	1	1	0

b)

	1	2	3	4	5	6
1	0	0	1	1	0	0
2	0	0	1	1	0	1
3	1	1	0	1	1	0
4	1	1	1	0	0	1
5	0	0	1	0	0	1
6	0	1	0	1	1	0

c)

	1	2	3	4	5	6
1	1	1	1	0	0	0
2	1	1	1	1	0	0
3	1	1	1	1	1	1
4	0	1	1	1	0	1
5	0	0	1	0	1	1
6	0	0	1	1	1	1

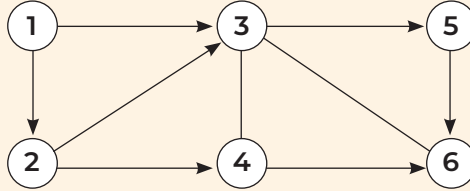
d)

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	1	0	1	1	0	0
3	1	1	0	1	1	1
4	1	1	1	0	1	1
5	0	0	1	1	0	1
6	0	0	1	1	1	0

e)

	1	2	3	4	5	6
1	0	1	0	0	1	0
2	1	0	1	1	0	0
3	0	1	0	1	1	1
4	0	1	1	0	0	0
5	1	0	1	0	0	1
6	0	0	1	0	1	0

2. Aşağıda verilen çizgeye ait komşuluk matrisi gösterimi hangi seçenekte doğru verilmiştir?



a)

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	1	1	0	0
3	0	0	0	1	1	1
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	0	0

b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	1	1	0	1
3	0	0	0	1	1	1
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	1	0	0	0

c)

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	1	1	0	0
3	0	0	0	1	1	1
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	1	1	1	0

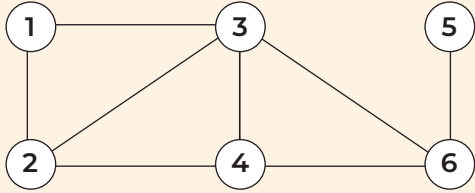
d)

	1	2	3	4	5	6
1	0	1	1	0	0	0
2	0	0	1	1	0	0
3	1	1	1	0	0	0
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	0	0

e)

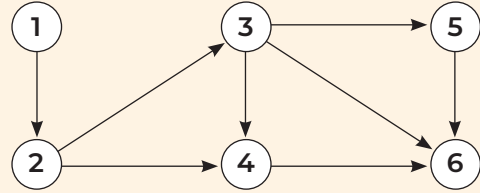
	1	2	3	4	5	6
1	0	1	0	0	1	0
2	0	0	1	1	0	0
3	0	1	0	1	0	1
4	0	0	1	0	1	0
5	0	0	0	0	0	1
6	0	0	0	0	0	0

3. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce genişlik arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, çizgedeki düğümleri ziyaret etmenin doğru sırası hangi seçenekte verilmiştir?



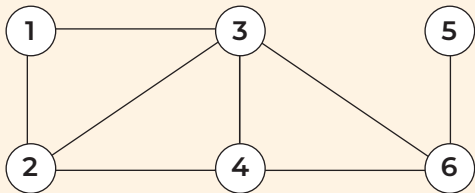
- a) 1, 2, 3, 4, 5, 6
- b) 1, 2, 3, 4, 6, 5
- c) 1, 2, 4, 3, 6, 5
- d) 1, 2, 4, 3, 5, 6
- e) 1, 2, 3, 6, 4, 5

5. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce genişlik arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, çizgedeki düğümleri ziyaret etmenin doğru sırası hangi seçenekte verilmiştir?



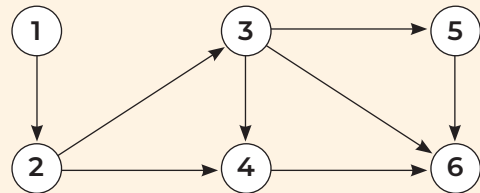
- a) 1, 2, 3, 4, 5, 6
- b) 1, 2, 3, 4, 6, 5
- c) 1, 2, 4, 3, 6, 5
- d) 1, 2, 4, 3, 5, 6
- e) 1, 2, 3, 6, 4, 5

4. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce derinlik arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, çizgedeki düğümleri ziyaret etmenin doğru sırası hangi seçenekte verilmiştir?



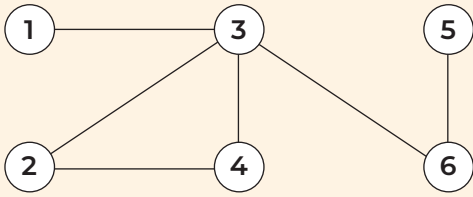
- a) 1, 2, 3, 4, 5, 6
- b) 1, 2, 3, 4, 6, 5
- c) 1, 2, 4, 3, 6, 5
- d) 1, 2, 4, 3, 5, 6
- e) 1, 2, 3, 6, 4, 5

6. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce derinlik arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, çizgedeki düğümleri ziyaret etmenin doğru sırası hangi seçenekte verilmiştir?



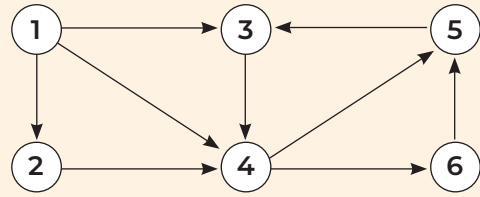
- a) 1, 2, 3, 4, 5, 6
- b) 1, 2, 3, 4, 6, 5
- c) 1, 2, 4, 3, 6, 5
- d) 1, 2, 4, 3, 5, 6
- e) 1, 2, 3, 6, 4, 5

7. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce genişlik arama algoritması çalıştırılacaktır. Bi-r düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, algoritma sonunda 6 düğümünün mesafe parametresinin alacağı değer aşağıdakilerden hangisi olacaktır?



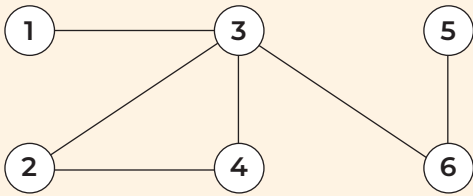
- a) 1
b) 2
c) 3
d) 4
e) 5

9. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce genişlik arama algoritması çalıştırılacaktır. Bi-r düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, algoritma sonunda 4 düğümünün ebeveyn parametresinin alacağı değer aşağıdakilerden hangisi olacaktır?



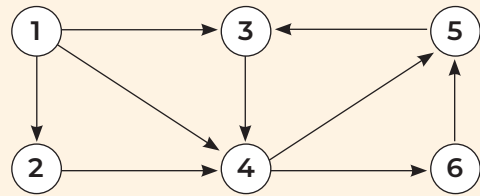
- a) 1
b) 2
c) 3
d) 5
e) 6

8. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce derinlik arama algoritması çalıştırılacaktır. Bi-r düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, algoritma sonunda 4 düğümünün bitiş zamanı aşağıdakilerden hangisi olacaktır?



- a) 3
b) 4
c) 5
d) 6
e) 7

10. Aşağıda verilen çizge üzerinde, başlangıç düğümü "1" olacak şekilde önce derinlik arama algoritması çalıştırılacaktır. Bi-r düğümün birden fazla komşusu olması durumunda komşular küçükten büyüğe doğru ziyaret edilecektir. Buna göre, algoritma sonunda 3 düğümünün başlangıç zamanı aşağıdakilerden hangisi olacaktır?



- a) 4
b) 5
c) 6
d) 7
e) 8

Yanıt Anahtarı: 1-A 2-A 3-B 4-B 5-A 6-B 7-B 8-C 9-A 10-B

Kaynakça

- Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.
- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.
- Çölkesen T.R. (2019) Veri Yapıları ve Algoritmalar, Papatya Yayıncılık.
- Sedgewick R., Wayne K. (2014) Algorithms, 4th Edition, Addison-Wesley
- Çamoğlu K. (2021) Algoritma, Kodlab.

BÖLÜM 11

ÇİZGE ALGORİTMALARI

ANAHTAR KAVRAMLAR

Minimum Kapsayan Ağaç, En Kısa Yol



ÖĞRENME HEDEFLERİ

- 1 Prim algoritmasını anlayacak ve uygulayabilecek
- 2 Kruskal algoritmasını anlayacak ve uygulayabilecek
- 3 Dijkstra algoritmasını anlayacak ve uygulayabilecek

GİRİŞ

Çizge algoritmaları, çizge veri yapısı ile modellenen bir çok problemi çözmek için kullanılır. Bu algoritmalar, ağ yönlendirmesinden zamanlamaya kadar bilgisayar bilimindeki çeşitli sorunları modellemek ve çözmek için kullanılır. En sık kullanılan çizge algoritmalarından ikisi, bir çizge için minimum kapsayan ağacı bulma problemi için geliştirilmiş Prim ve Kruskal algoritmalarıdır. Algoritmalar ağırlıklı yönlü ve yönsüz çizgelerle kullanılmak üzere tasarlanmış ve bir çizgedeki tüm düğümleri en az maliyetle birbirlerine bağlamanın en verimli yolunu bulmak için kullanılmaktadırlar. Prim ve Kruskal algoritmaları, ağ tasarımı, kümeleme ve veri madenciliği gibi çeşitli uygulamalarda kullanılmıştır. Bunlara ek olarak, bir çizgedeki iki düğüm arasındaki en kısa yolu bulmak için Dijkstra en kısa yol algoritması tercih edilir. Bu algoritma, ağ yönlendirme, ağ optimizasyonu ve iki nokta arasındaki en kısa mesafenin gerekli olduğu diğer uygulamalarda da kullanılmaktadır. Ayrıca Dijkstra algoritması, iki şehir arasındaki en kısa yolu bulmaktan bir paket teslimat hizmeti için en verimli rotayı bulmaya kadar çok çeşitli problemleri çözmek için kullanılmıştır.



1. MİNİMUM KAPSAYAN AĞAÇ

Minimum Kapsayan Ağaç problemi, verilen bir çizgedeki tüm düğümleri minimum toplam kenar ağırlığıyla birleştiren kenarların alt kümesini

bulmak şeklinde tanımlanmaktadır. Bu problemin çözümü için yaygın olarak iki algoritma kullanılmaktadır: Prim ve Kruskal algoritmaları.

1.1. Prim Algoritması

Minimum kapsayan ağacı bulmak için en sık kullanılan algoritma Prim algoritmasıdır. Prim algoritması, çizgede bir düğüm noktasını başlangıç noktası seçerek çalışmaktadır. Sonrasında algoritma başlangıç düğümünün komşularından

başlayarak sırasıyla bütün düğümleri kapsayan ağaca ekleyebileceği en az ağırlıklı kenarları kapsayan ağaca ekleyerek sonuçta döneceği minimum kapsayan ağacı oluşturmaktadır.

```

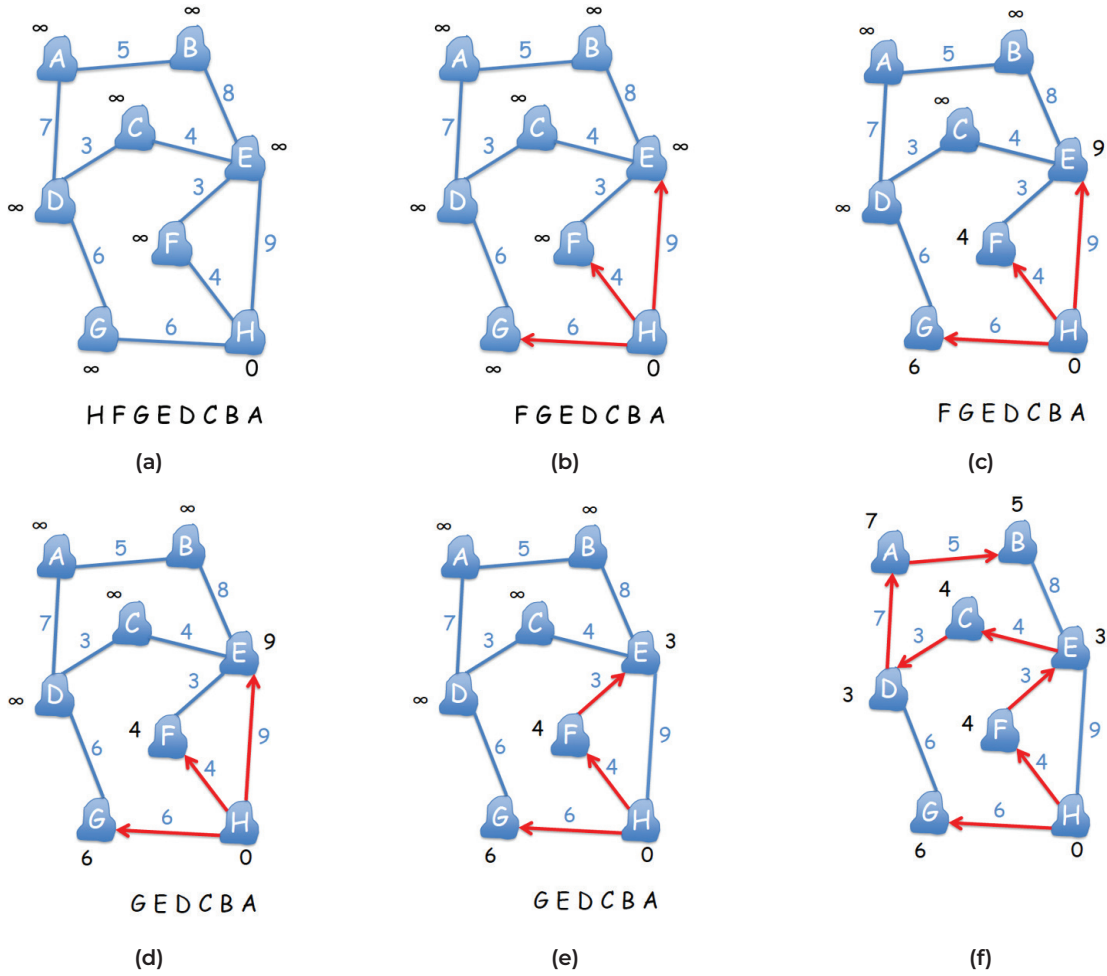
MST-PRIM (G, w, r)
for each vertex  $u \in G.V$ 
     $u.key = \infty$ 
     $u.par = NIL$ 
 $r.key = 0$ 
 $Q = \emptyset$ 
for each vertex  $u \in G.V$ 
    INSERT (Q, u)
while  $Q \neq \emptyset$ 
     $u = \text{EXTRACT-MIN}(Q)$ 
    for each vertex  $v$  in  $G.Adj[u]$ 
        if  $v \in Q$  and  $w(u, v) < v.key$ 
             $v.par = u$ 
             $v.key = w(u, v)$ 
    DECREASE-KEY (Q, v,  $w(u, v)$ )

```

Şekil 11.1 Prim algoritmasına ait sözde kod

Prim algoritması bir çizge, ağırlık matrisi ve başlangıç düğümünü girdi olarak almaktadır. Algoritma ilk olarak çizgedeki bütün düğümlerin anahtar değerlerini sonsuz ve ebeveyni null olarak belirler. Daha sonra başlangıç düğümünün anahtar değerine 0 atar. Algoritma boş minimum öncelikli bir Q kuyruğu oluşturur ve bütün düğümleri kuyruğa anahtar değerlerine göre yerleştirir. Sonrasında algoritma, bir döngü içerisinde Q kuyruğu boş olmadığı sürece en küçük anahtar değerine sahip düğümü (bu düğümü u diyelim) kuyruktan çıkarır, ve u düğümünün

komşularının ebeveyn ve anahtar parametrelerini eğer komşular hala kuyruktaysa ve komşu ile u düğümünü birleştiren kenarın ağırlığı komşunun anahtar değerinden küçükse yeniler. Bu şekilde algoritma esnasında, herhangi bir düğümü kapsayan ağaca bağlayabilecek daha az ağırlıklı bir kenar keşfedilmişse, ilgili düğümün yeni keşfedilen kenar üzerinden ağaca eklenmesi; ve algoritma çalıştırılması esnasında olası bütün kenarları taradığından çizgenin herbir düğümünün en az ağırlıklı kenar üzerinden kapsayan ağaca bağlanması sağlanmaktadır.



Şekil 11.2 üzerinden Prim algoritmasının verilen bir çizge için minimum kapsayan ağacı nasıl oluşturduğu adım adım gösterilecektir:

Şekil 11.2 Prim algoritmasının örnek bir çizge üzerinde çalışma adımları

(a) Algoritma ilk olarak bütün düğümlerin anahtar değerlerine sonsuz değeri atar ve başlangıç düğümünün anahtar değerini 0 olarak günceller.

(b) Anahtar değeri en küçük olan düğümü (H düğümü) Q kuyruğundan çıkartır ve bu düğümün komşuları E, F ve G düğümlerinin hepsi hala Q kuyruğunda olduğundan ve bu komşu düğümlerin anahtar değerleri onları H düğümü ile birleştiren kenarın ağırlığından küçük olduğundan, bu komşu düğümlerin ebeveyn parametresini H düğümü olarak günceller.

(c) Ebeveyn parametresini H düğümü olarak güncellediği E, F ve G düğümlerinin anahtar değerlerini sırasıyla 9, 4 ve 6 olarak günceller ve Q kuyruğunu en küçük anahtar değere sahip düğüm

en ulaşılabilir konumda olacak şekilde günceller.

(d) Kalan elemanlardan en küçük anahtar değerine sahip düğümü (F düğümü) Q kuyruğundan çıkartır.

(e) F düğümünün komşularından E düğümü hala Q kuyruğunda olduğundan ve F'yi E ile birleştiren kenarın ağırlığı E'nin anahtarının güncel değerinden küçük olduğundan, E düğümünün ebeveyn parametresini F düğümü olarak ve E'nin anahtar değerini 3 olarak günceller. Burada da E düğümünde gözlemlendiği üzere, algoritma E düğümünü finalde oluşturacağı minimum kapsayan ağaca bağlayacak daha az ağırlıklı bir kenar keşfettiğinde ilgili düğümü yeni keşfettiği kenar üzerinden ağaca bağlamaktadır.

(f) Algoritma benzer adımları diğer düğümlere de uygulayarak finalde minimum kapsayan

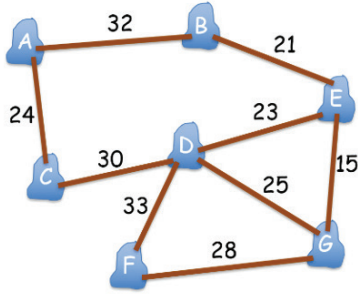
ağacı oluşturmaktadır. Şekilde kırmızı ile gösterilen düğümlerin ebeveyn parametreleri, ilgili

düğümün hangi kenar yoluyla minimum kapsayan ağaca bağlanacağını belirtmektedir.

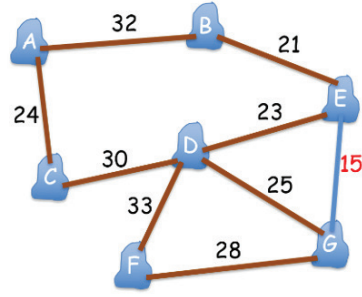
1.2. Kruskal Algoritması

Kruskal algoritması, minimum kapsayan ağacı bulmak için tercih edilen başka bir algoritmadır. Bu algoritma, çizgedeki kenarları ağırlığa göre sıralayarak ve ardından bunları birer birer kapsayan ağaca ekleyerek çalışır. Kapsayan ağaca

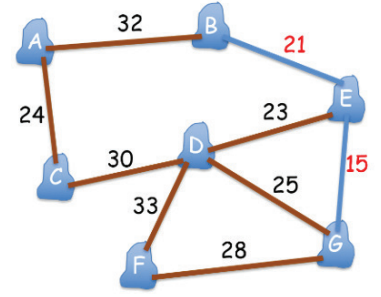
bir kenar eklenmeden önce, ilgili kenarın ağaca eklenmesi ile döngü olup oluşmadığı kontrol edilmelidir. Döngü oluşuyorsa, ilgili kenar ağaca eklenmez ve bir sonraki en az ağırlıklı kenar ile ağaç oluşum süreci devam ettirilir.



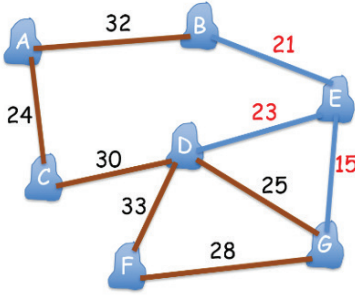
(a)



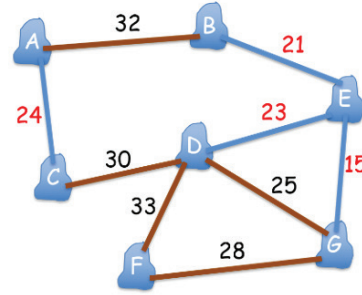
(b)



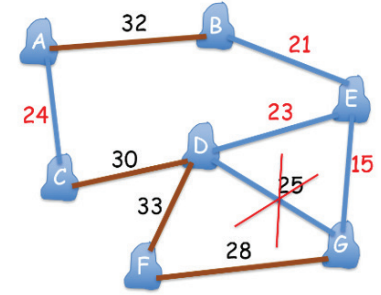
(c)



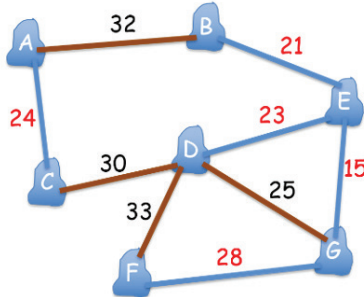
(d)



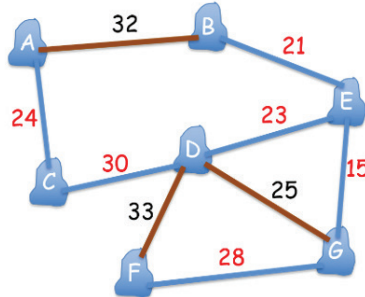
(e)



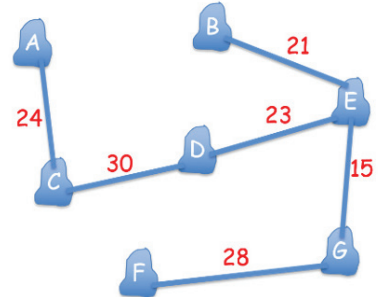
(f)



(g)



(h)



(i)

Şekil 11.3 Kruskal algoritmasının örnek bir çizge üzerinde çalışma adımları

Şekil 11.3 üzerinden Kruskal algoritmasının verilen bir çizge için minimum kapsayan ağacı nasıl oluşturduğu adım adım gösterilecektir:

(a) Düşümler ve kenar ağırlıkları örnek çizge üzerinde gösterilmiştir.

(b) En küçük ağırlığa sahip kenar (E, G) olduğu için ağaca eklenir.

(c) İkinci en küçük ağırlık değeri 21 olduğu için (B, E) kenarı döngü oluşturmadığı kontrol edilerek ağaca eklenir.

(d) (D, E) kenarı ağaca eklenir.

(e) (A, C) kenarı kalan kenarlar içerisinde en küçük olduğu ve döngü oluşturmadığı için ağaca eklenir.

(f) (D, G) kenarı kalan kenarlar içerisinde en küçük ağırlığa sahip olmasına rağmen eklenmesi durumunda döngü oluşturacağı için ağaca eklenmeden sıradaki kenar ile devam edilir.

(g) (F, G) kenarı kalan kenarlar içerisinde en küçük ağırlığa sahip olduğu için ve döngü oluşturmadığı için ağaca eklenir.

(h) En son olarak (C, D) kenarı eklenir. Dikkat edilirse n düğümlü bir çizge için; $n - 1$ döngü oluşturmayan en az ağırlıklı kenar, minimum kapsayan ağaç oluşturmak için yeterli olacaktır.

(i) Kruskal algoritmasının oluşturduğu minimum kapsayan ağaç.

2. EN KISA YOL

Tek kaynaklı en kısa yol algoritmaları, belirli bir kaynak düğüm ile bir çizgedeki diğer tüm düğümler arasındaki en kısa yolu bulmak için kullanılır. Bu algoritmalar, harita üzerindeki iki konum arasındaki en verimli rotayı bulmak, bir

bilgisayar ağındaki trafiği yönlendirmek veya diğer birçok optimizasyon problemlerini çözmek için tercih edilir. En popüler tek kaynaklı en kısa yol algoritmaları Dijkstra, Bellman-Ford ve Floyd-Warshall'dır.

2.1. Dijkstra Algoritması

Dijkstra algoritması, belirli bir başlangıç düğümünden mesafe değerlerine göre çizgenin diğer tüm düğümlerine giden en kısa yolları bulur. İlk olarak, başlangıçtan kendisine en yakın düğümüne, ardından en yakın ikinci düğüme giden en kısa yolu bulur. Tüm düğümler ziyaret edilene

kadar başlangıç düğümünden çizgedeki diğer tüm düğümlere giden en kısa yolları art arda hesaplayarak çalışır. Bu algoritma ancak çizgedeki tüm kenar negatif olmayan ağırlıklara sahip olduğunda doğru sonuç üretebilmektedir.

Dijkstra(G,s)

```

for each u of V
    u.dis =  $\infty$ 
    u.par = nil
s.dis = 0
initialize an empty set S
create a minimum priority Q on V
while Q  $\neq$  { }
    u = ExtractMin(Q)
    S = S  $\cup$  {u}
    for each v of Adj(u)
        if v.dis > u.dis + w(u,v)
            v.dis = u.dis + w(u,v)
            v.par = u
    update Q

```

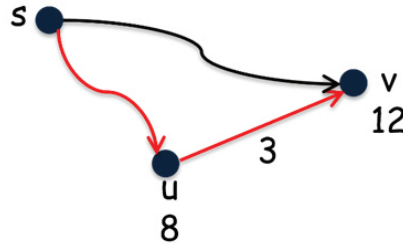
Şekil 11.4 Dijkstra algoritmasına ait sözde kod

Sözde kod ile gösterimi Şekil 11.4'te verilen Dijkstra algoritması bir çizgeyi ve başlangıç düğümünü parametre olarak alır. Algoritma öncelikli olarak, çizgenin bütün düğümlerinin mesafe ve ebeveyn parametrelerine başlangıç değerlerini atar. Burada mesafe değerinin başlangıç değeri sonsuz ve ebeveyn parametresinin başlangıç değeri olarak null'dur. Başlangıç düğümünün mesafe değeri 0 olarak güncellenir ve algoritmanın bu düğümden başlaması sağlanır. S boş kümesi

ve bütün düğümlerin mesafe değerlerine bağlı olmak üzere minimum öncelikli bir Q kuyruğu oluşturulur. Bir döngü içerisinde Q kuyruğu boş olmadığı sürece kuyruktaki en küçük mesafe değerine sahip düğüm çıkarılır, S kümesine eklenir ve her bir komşusu için gevşetme işlemi uygulanır. Gevşetme işleminin amacı, başlangıç düğümünden diğer düğümlere daha kısa uzunluklu bir yol keşfedildiğinde, ilgili düğüme giden en kısa yolu keşfedilen yola göre güncellemektir.

Relax(u, v)

```
if v.dis > u.dis + w(u,v)
    v.dis = u.dis + w(u,v)
    v.par = u
```

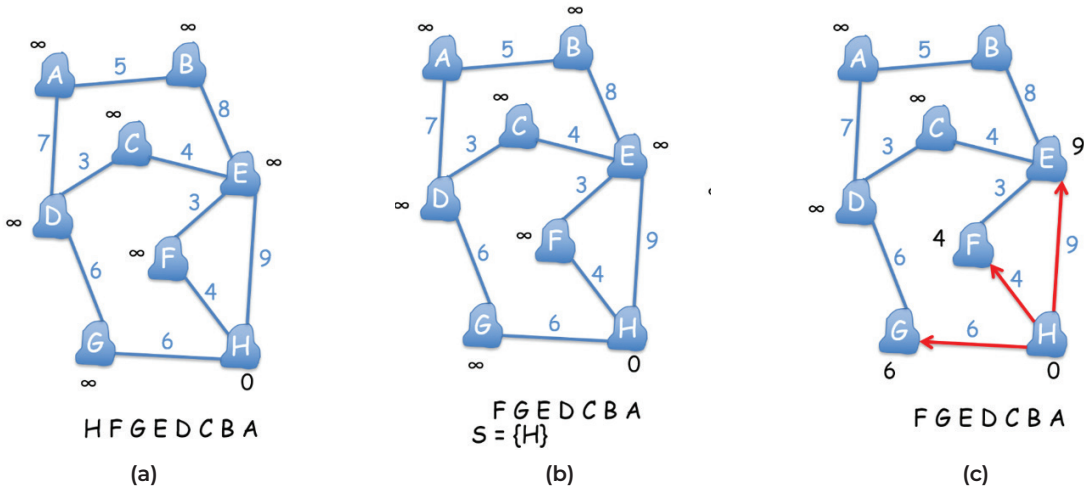


$v.dis > u.dis + w(u,v)$ $\rightarrow$ $v.dis = u.dis + w(u,v)$
 $12 > 8 + 3$ $\rightarrow$ $v.dis = 11$
 $v.par = u$

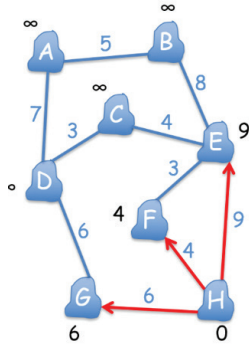
Şekil 11.5 Gevşetme işlemini ait bir örnek

Şekil 11.5'te gösterildiği gibi s düğümünden v düğümüne olan en kısa mesafe değerinin 12 olması durumunda, v düğümüne u düğümü üzerinden gidilmesi daha az maliyetli olacağı için v düğümünün mesafe değeri gevşetilerek 11 olarak

belirlenecektir. Şekil 11.6 üzerinden Dijkstra algoritmasının H düğümünden diğer düğümlere en kısa yolları nasıl bulduğu adım adım gösterilecektir:

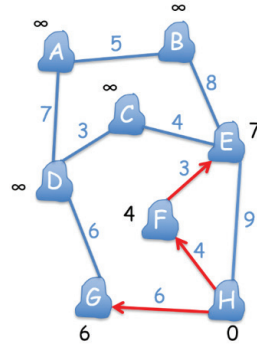


Şekil 11.6 Dijkstra algoritmasının örnek bir çizge üzerinde çalışma adımları



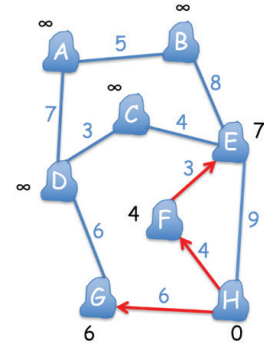
G E D C B A
S = {H, F}

(d)



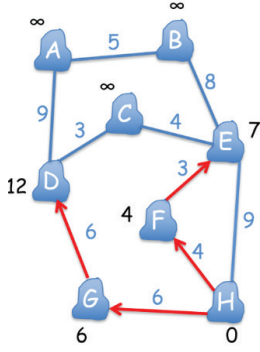
G E D C B A
S = {H, F}

(e)



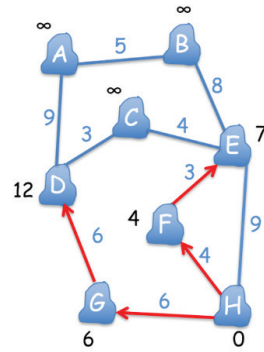
E D C B A
S = {H, F, G}

(f)



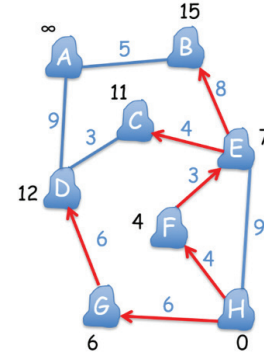
E D C B A
S = {H, F, G}

(g)



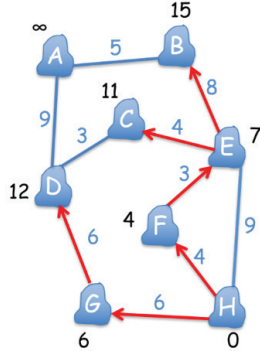
D C B A
S = {H, F, G, E}

(h)



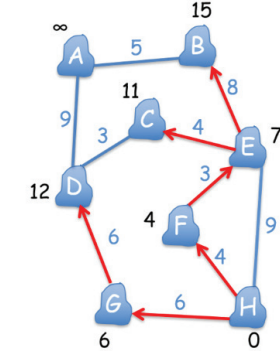
D C B A

(i)



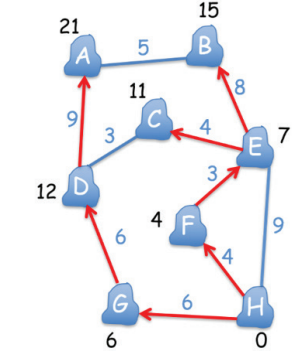
D B A
S = {H, F, G, E, C}

(j)



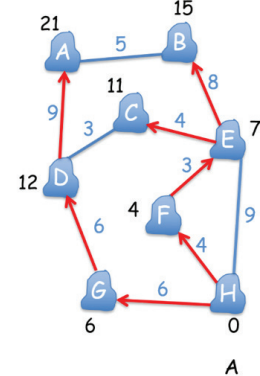
B A
S = {H, F, G, E, C}

(k)

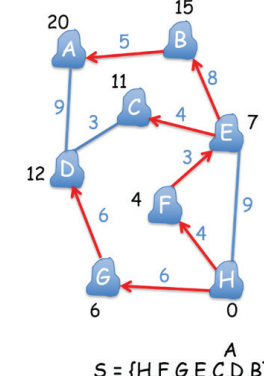


B A
S = {H, F, G, E, C, D}

(l)

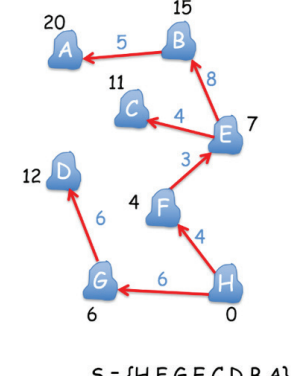


(m)



A
S = {H, F, G, E, C, D, B}

(n)



S = {H, F, G, E, C, D, B, A}

(o)

(a) Algoritma ilk olarak bütün düğümlere sonsuz mesafe değeri atar ve ardından başlangıç düğümünün mesafesini 0 olarak günceller ve çalışmaya başlar.

(b) En küçük mesafe değerli H düğümü Q kuyruğundan çıkartılarak S kümesine eklenir.

(c) H düğümünün komşuları E, F ve G düğümlerine gevşetme işlemi uygulanır: düğümlerin mesafe değerleri sırasıyla 9, 4, ve 6 olarak ve ebeveyn parametreleri H düğümü olarak güncellenir. Q minimum öncelikli kuyruğu düğümlerin yeni mesafe değerlerine göre güncellenir.

(d) Q kuyruğundaki en kısa mesafe değerine sahip F düğümü kuyruktan çıkartılarak S kümesine eklenir.

(e) F düğümünün komşusu E düğümüne gevşetme işlemi uygulanır: mesafe değeri 9 iken gevşetilmesiyle yeni mesafe değeri olarak 7 belirlenir ve ebeveyn parametresi F düğümü olarak güncellenir.

(f) Q kuyruğundaki en kısa mesafe değeri G düğümüne ait olduğu için bu düğüm kuyruktan çıkartılır ve S kümesine eklenir.

(g) G düğümünün komşusu D düğümüne gevşetme işlemi uygulanır: mesafe değeri 12 ve ebeveyn parametresi G düğümü olarak güncellenir.

(h) Q kuyruğundaki en kısa mesafe değeri E düğümüne ait olduğu için bu düğüm kuyruktan çıkartılır ve S kümesine eklenir.

(i) E düğümünün komşuları B ve C düğümlerine gevşetme işlemi uygulanır: düğümlerin me-

safe değerleri sırasıyla 15 ve 11 olarak ve ebeveyn parametreleri E düğümü olarak güncellenir. Q minimum öncelikli kuyruğu düğümlerin yeni mesafe değerlerine göre güncellenir.

(j) Q kuyruğundaki en kısa mesafe değerine sahip C düğümü kuyruktan çıkartılarak S kümesine eklenir.

(k) C düğümünün komşusu G düğümünün anahtar değeri 12, C'nin anahtar değeri ile (C, E) kenarının ağırlığı toplamından küçük olduğundan G düğümüne gevşetme işlemi uygulanmaz.

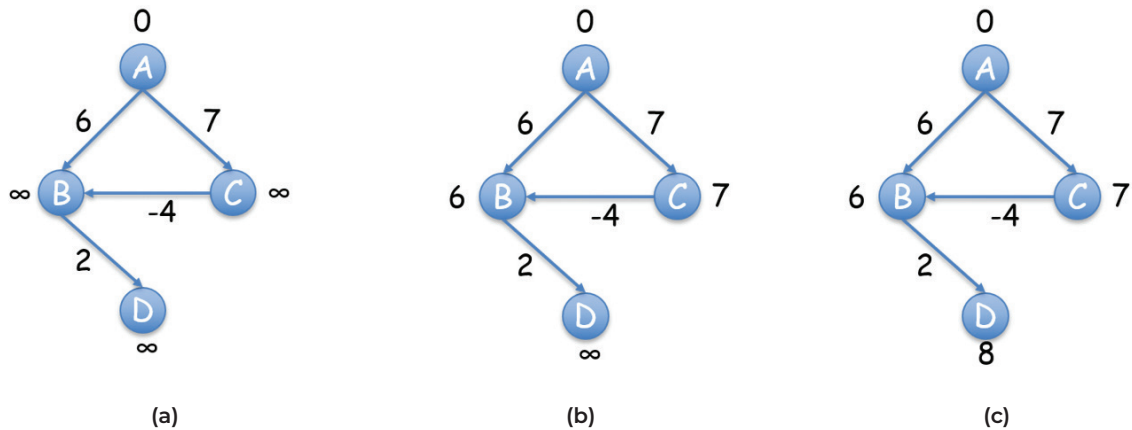
(l) Q kuyruğundaki en kısa mesafe değerine sahip D düğümü kuyruktan çıkartılarak S kümesine eklenir. D düğümünün komşusu A düğümüne gevşetme işlemi uygulanır: ebeveyn parametresi D düğümü ve anahtar değeri 21 olarak güncellenir.

(m) Q kuyruğundaki en kısa mesafe değerine sahip B düğümü kuyruktan çıkartılarak S kümesine eklenir.

(n) B düğümünün komşusu A düğümüne gevşetme işlemi uygulanır: ebeveyn parametresi B düğümü ve anahtar değeri 20 olarak güncellenir.

(o) Q kuyruğundaki en kısa mesafe değerine sahip A düğümü kuyruktan çıkartılarak S kümesine eklenir ve nihayetinde ebeveyn parametreleri kullanılarak en kısa yol ağacı oluşturulur.

Yukarıda da belirtildiği gibi Dijkstra algoritması negatif değerlere sahip bir çizge üzerinde doğru en kısa yolları bulamayacaktır. Burada bu durum bir örnek üzerinden anlatılacaktır.



Şekil 11.7 Negatif ağırlıklı bir kenar olması durumunda Dijkstra algoritmasının doğru sonuç üretmediğini gösteren örnek

A düğümünü kaynak düğüm olarak kabul edilmesi durumunda Dijkstra algoritması diğer düğümlere olan en kısa mesafeyi bulacaktır. Dijkstra algoritmasının çalıştırılması sonunda elde edilen mesafe değerleri Şekil 11.7.c'de gös-

terilmiştir. D düğümüne C düğümü üzerinden gidilmesi durumunda gerçek en kısa mesafe 5 olacaktır. Örnekten anlaşılacağı üzere Dijkstra D algoritması negatif kenar olması durumunda doğru sonuç üretememektedir.

BÖLÜM ÖZETİ

Bu bölüm, çizge algoritmalarını ele almakta olup, minimum kapsayan ağaç ve en kısa yol problemlerini çözmek için kullanılan temel algoritmaları incelemektedir. Çizge algoritmaları, ağ yönlendirmesi, zamanlama, veri madenciliği ve optimizasyon gibi birçok alanda kullanılmaktadır. Bu bağlamda en yaygın kullanılan algoritmalar Prim, Kruskal ve Dijkstra algoritmalarıdır.

Bir çizgede bulunan tüm düğümleri en düşük toplam kenar ağırlığıyla bağlayan en küçük ağacı oluşturan algoritmalar minimum kapsayan ağaç algoritmaları olarak adlandırılır. Prim ve Kruskal algoritmaları, bu problemi çözmek için kullanılan en yaygın yöntemlerdir.

Prim Algoritması, bir başlangıç düğümünden yola çıkarak her adımda en düşük maliyetli kenarı seçerek ağacı oluşturur. Algoritma, çizgedeki düğümleri birer birer ekleyerek her zaman en düşük ağırlıklı bağlantıyı tercih eder. Özellikle yoğun bağlantılara sahip çizgelerde daha verimli çalışmaktadır.

Kruskal Algoritması, kenarları ağırlıklarına göre sıralayarak en düşük ağırlıklı kenardan başlayıp minimum kapsayan ağacı oluşturmaktadır. Kruskal algoritmasının temel farkı, kenarları ağırlıklarına göre sıralı bir şekilde işleyerek döngü oluşturmada ağacı tamamlamasıdır. Özellikle seyrek çizgelerde daha etkilidir.

Çizgelerdeki iki düğüm arasındaki en kısa mesafeyi bulmaya yönelik kullanılan algoritmalar en kısa yol algoritmalarıdır. Bu algoritmalar genellikle ulaşım, ağ trafiği yönlendirme ve optimizasyon problemlerinde kullanılır.

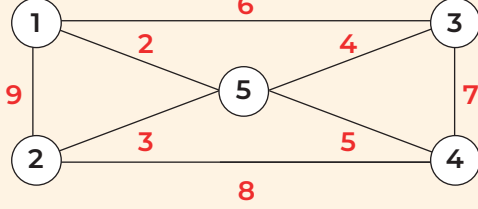
Dijkstra Algoritması, belirli bir başlangıç düğümünden diğer tüm düğümlere olan en kısa yolları bulmak için kullanılır. Algoritma, başlangıç düğümünden itibaren en kısa yolu bulmaya çalışarak ilerler ve her adımda en düşük mesafeye sahip düğümü işleyerek nihai en kısa yolu belirler. Negatif ağırlıklı kenarlarda doğru çalışmaz; bu yüzden ağırlıkların negatif olmadığı çizgelerde kullanılır.

Dijkstra algoritması, öncelikli kuyruk yapısını kullanarak, en düşük maliyetli düğümü işleyerek ilerler. Algoritma, belirli bir noktadan hareket ederek her düğüme en kısa yolu belirleyip, en kısa yol ağaçlarını oluşturur. Bunun sonucunda, belirlenen düğüme en düşük maliyetle ulaşmak mümkün hale gelir.

Bu bölümde, Prim ve Kruskal algoritmalarının minimum kapsayan ağaç problemlerine, Dijkstra algoritmasının ise en kısa yol problemlerine nasıl uygulandığı incelenmiştir. Prim algoritması, ağaç yapısını oluştururken düğümleri merkeze alarak en düşük maliyetli kenarları eklerken, Kruskal algoritması kenarları sıralayarak döngü oluşturmada ilerlemektedir. Dijkstra algoritması ise belirli bir düğümden hareketle en kısa yolları hesaplamak için kullanılır. Bu üç algoritma, ulaşım, ağ yönlendirme, optimizasyon ve veri madenciliği gibi alanlarda önemli uygulamalara sahiptir.

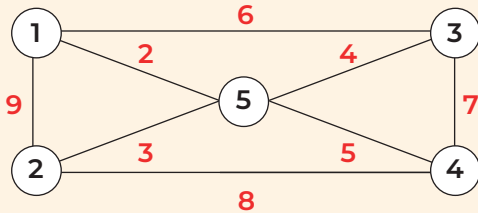
GÖZDEN GEÇİRELİM

1. Aşağıda verilen çizgeye ait Minimum Kapsayan ağaçta hangi kenar bulunmaz?



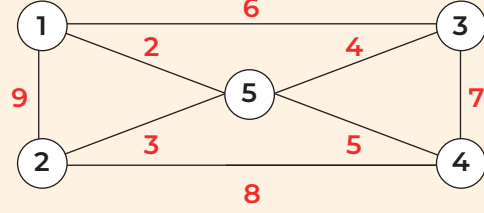
- a) (1, 5)
- b) (2, 5)
- c) (3, 5)
- d) (4, 5)
- e) (1, 3)

2. Aşağıda verilen çizge üzerinde Kruskal Minimum Kapsayan ağaç algoritması çalıştırılacaktır. Algoritma ilk olarak hangi kenardan çalışmaya başlar?



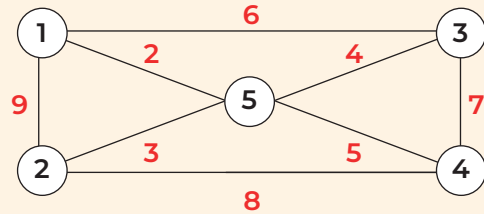
- a) (1, 5)
- b) (2, 4)
- c) (3, 4)
- d) (2, 5)
- e) (1, 3)

3. Aşağıda verilen çizge üzerinde Prim Minimum Kapsayan ağaç algoritması "5" düğümü seçilerek başlatılması durumunda, algoritma sonucunda "4" düğümünün anahtar parametresinin alacağı değer ne olacaktır?



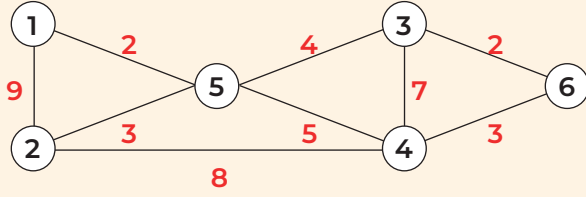
- a) 4
- b) 5
- c) 6
- d) 7
- e) 8

4. Aşağıda verilen çizgeye ait Minimum Kapsayan ağaçta kaç adet kenar bulunmalıdır?



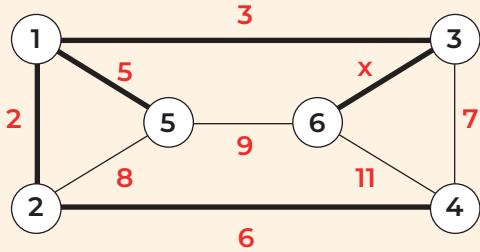
- a) 3
- b) 4
- c) 5
- d) 6
- e) 7

5. Aşağıda verilen çizgeye ait Minimum Kapsayan ağaçta hangi kenar bulunmaz?



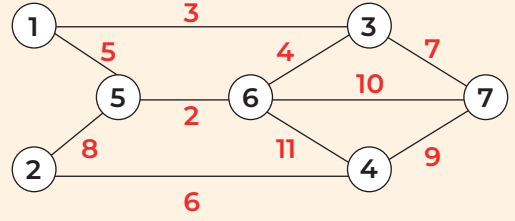
- a) 4
- b) 6
- c) 8
- d) 10
- e) 12

6. Aşağıda verilen çizgede kalın kenarlar Minimum Kapsayan Ağacı oluşturmaktadır. Bu durumda "x" ile ifade edilen kenar ağırlığı aşağıdakilerden hangisi olamaz.



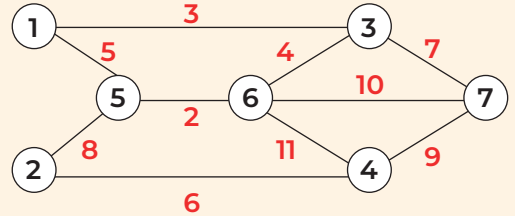
- a) 2
- b) 4
- c) 6
- d) 8
- e) 10

7. Aşağıda verilen çizgeye ait Minimum Kapsayan ağaçta hangi kenar bulunmaz?



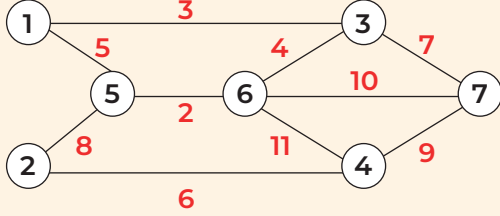
- a) (1, 5)
- b) (1, 3)
- c) (5, 6)
- d) (3, 7)
- e) (2, 4)

8. Aşağıda verilen çizge üzerinde Kruskal Minimum Kapsayan ağaç algoritması çalıştırılacaktır. Algoritma ilk olarak hangi kenardan çalışmaya başlar?



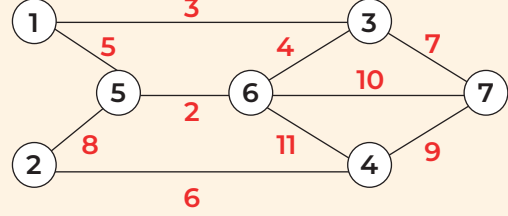
- a) (1, 5)
- b) (2, 4)
- c) (2, 5)
- d) (3, 6)
- e) (5, 6)

9. Aşağıda verilen çizge üzerinde Prim Minimum Kapsayan ağaç algoritması “6” düğümü seçilerek başlatılması durumunda, algoritma sonucunda “1” düğümünün anahtar değeri ne olacaktır?



- a) 3
- b) 4
- c) 5
- d) 7
- e) 9

10. Aşağıda verilen çizgeye ait Minimum Kapsayan ağaçta kaç adet kenar bulunmalıdır?



- a) 3
- b) 4
- c) 5
- d) 6
- e) 7

Yanıt Anahtarı: 1-E 2-A 3-B 4-B 5-C 6-E 7-A 8-E 9-A 10-D

Kaynakça

Levitin A. (2012) Introduction to The Design & Analysis of Algorithms, 3rd Edition, Pearson.

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) Introduction to Algorithms, 3rd Edition, MIT Press.

Çölkesen T.R. (2019) Veri Yapıları ve Algoritmalar, Papatya Yayıncılık

Sedgewick R., Wayne K. (2014) Algorithms, 4th Edition, Addison-Wesley

Çamoğlu K. (2021) Algoritma, Kodlab