



a n a d o l u m
e K a m p ü s
ve
a n a d o l u m o b i l
dilediğin yerden,
dilediğin zaman,
öğrenme fırsatı!



(ekampus.anadolu.edu.tr)



(mobil.anadolu.edu.tr)

ekampus.anadolu.edu.tr



Açıköğretim Destek Sistemi
Açıköğretim Sistemi ile ilgili

merak ettiğiniz her şey AOS Destek Sisteminde...

-  Kolay Soru Sorma ve Soru-Yanıt Takibi
-  Sıkça Sorulan Sorular ve Yanıtları
-  Canlı Destek (Hafta İçi Her Gün)
-  Telefonla Destek

aosdestek.anadolu.edu.tr

AOS DESTEK Sistemi İletişim ve Çözüm Masası

0850 200 46 10

www.anadolu.edu.tr

T.C. ANADOLU ÜNİVERSİTESİ YAYINI NO: 3445
AÇIKÖĞRETİM FAKÜLTESİ YAYINI NO: 2293

ALGORİTMALAR VE PROGRAMLAMA

Yazarlar

Arş.Gör.Dr. Burcu YILMAZEL (Ünite 1, 2, 3)

Dr.Öğr.Üyesi Sevcan YILMAZ GÜNDÜZ (Ünite 4, 5)

Dr.Öğr.Üyesi Alper Kürşat UYSAL (Ünite 6, 7, 8)

Editör

Prof.Dr. Serkan GÜNAL

Bu kitabın basım, yayım ve satış hakları Anadolu Üniversitesi'ne aittir.
“Uzaktan Öğretim” teknigine uygun olarak hazırlanan bu kitabı bütün hakları saklıdır.
İlgili kuruluştan izin alınmadan kitabı tümü ya da bölmeleri mekanik, elektronik, fotokopi, manyetik kayıt
veya başka şekillerde çoğaltılamaz, basılamaz ve dağıtılamaz.

Copyright © 2016 by Anadolu University
All rights reserved

No part of this book may be reproduced or stored in a retrieval system, or transmitted
in any form or by any means mechanical, electronic, photocopy, magnetic tape or otherwise, without
permission in writing from the University.

Öğretim Tasarımcısı
Prof.Dr. Tevfik Volkan Yüzer

Grafik Tasarım Yönetmenleri
*Prof. Tevfik Fikret Uçar
Doç.Dr. Nilgün Salur
Öğr.Gör. Cemalettin Yıldız*

Kapak Düzeni
Prof.Dr. Halit Turgay Ünalan

Grafikerler
*Ayşegül Dibek
Özlem Çayırlı
Hilal Özcan*

Dizgi ve Yayıma Hazırlama
Kitap Hazırlama Grubu

Algoritmalar ve Programlama

E-ISBN
978-975-06-3084-2

Bu kitabı tüm hakları Anadolu Üniversitesi'ne aittir.
ESKİŞEHİR, Ocak 2019
3107-0-0-1709-V01

İçindekiler

Önsöz vii

Algoritma Kavramı ve Programlama Temelleri	2	1. ÜNİTE
GİRİŞ	3	
ALGORİTMA NEDİR?	3	
Algoritmaların Temel Özellikleri	4	
ALGORİTMA GÖSTERİM YÖNTEMLERİ	4	
Konuşma Dili	5	
Akiş Şeması	5	
Sözde Kod	6	
ALGORİTMALARIN SINIFLANDIRILMASI	7	
Özyinelemeli Algoritmalar (Simple Recursive Algorithms)	7	
Geri İzlemeli Algoritmalar (Backtracking Algorithms)	8	
Böl ve Yönet Algoritmaları (Divide and Conquer Algorithms)	8	
Dinamik Programlama (Dynamic Programming)	9	
Açgözlü Algoritmalar (Greedy Algorithms)	10	
Kaba Kuvvet Algoritmaları (Brute Force Algorithms)	10	
VERİ YAPILARI	10	
Özet	12	
Kendimizi Sınayalım	13	
Kendimizi Sınayalım Yanıt Anahtarları	14	
Sıra Sizde Yanıt Anahtarları	14	
Yararlanılan ve Başvurulabilecek Kaynaklar	15	
Diziler, Bağlı Listeler, Kuyruklar ve Yığınlar	16	2. ÜNİTE
GİRİŞ	17	
DİZİLER	17	
Dizilerin Tanımlanması	18	
Dizilere Değer Atama	18	
Çok Boyutlu Diziler	20	
İki Boyutlu Diziler	20	
Üç Boyutlu Diziler	21	
BAĞLI LİSTELER	22	
Bağılı Listeler ile Dizilerin Karşılaştırması	22	
Bağılı Liste Türleri	22	
Tek Yönlü Bağılı Liste	22	
Çift Yönlü Bağılı Liste	23	
Dairesel Bağılı Liste	23	
Bağılı Listelerde Temel İşlemler	23	
Düğüm Yapısını Oluşturma	23	
Listeye Eleman Ekleme (Insertion)	24	
Listeden Eleman Çıkarma (Deletion)	24	
Listeyi Gezinme (Traversal)	24	
KUYRUKLAR	26	
Dizi ile Kuyruk Uygulaması	26	
Bağılı Liste ile Kuyruk Uygulaması	28	
YIĞINLAR	30	

Dizi ile Yiğin Uygulaması	31
Bağlı Liste ile Yiğin Uygulaması	32
Özet	35
Kendimizi Sınayalım	36
Kendimizi Sınayalım Yanıt Anahtarı	37
Sıra Sizde Yanıt Anahtarı	37
Yararlanılan ve Başvurulabilecek Kaynaklar	39

3. ÜNİTE**Ağaçlar, Yiğin Ağaçları ve Özetleme Tabloları.....** **40**

GİRİŞ	41
AĞAÇLAR	41
İkili Ağaçlar (Binary Trees)	42
İkili Ağacılarda Gezinme Yöntemleri	43
Preorder Gezinme	43
Inorder Gezinme	44
Postorder Gezinme	44
İkili Arama Ağaçları (Binary Search Trees)	44
İkili Arama Ağacına Düğüm Ekleme	45
İkili Arama Ağacından Düğüm Çıkarma	46
AVL Ağaçları	47
AVL Ağacına Düğüm Ekleme ve Denge Korunuμu	49
AVL Ağacından Düğüm Çıkarma ve Denge Korunuμu	50
YIĞIN AĞAÇLARI	51
Dizi ile Yiğin Ağacı Uygulaması	52
Yiğin Ağacında En Küçük Elemanı Elde Etme	52
Yiğin Ağacından En Küçük Elemanı Çıkarma: Aşağı Yönlendirme	52
Yiğin Ağacına Eleman Ekleme: Yukarı Yönlendirme	53
ÖZETLEME (HASH) TABLOLARI	53
Çatışmalar	54
Çatışma Çözüm Yöntemleri	55
Ayrık Zincirleme (Separate Chaining)	55
Açık Adresleme (Open Addressing)	56
Özet	57
Kendimizi Sınayalım	59
Kendimizi Sınayalım Yanıt Anahtarı	61
Sıra Sizde Yanıt Anahtarı	62
Yararlanılan ve Başvurulabilecek Kaynaklar	64

4. ÜNİTE**Algoritma Tasarımı** **65**

GİRİŞ	67
Hangi Problemler Algoritmalar ile Çözülebilir?	67
Teknolojik Olarak Algoritmalar	68
ALGORİTMA İLE PROBLEM ÇÖZME	68
ALGORİTMA TASARLAMA TEKNİKLERİ	70
Döngü-Tekrarlama Algoritmaları	70
Küçült-Fethet (Decrease&Conquer) Yöntemi ile Algoritma Tasarlama	73
Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet (Divide & Conquer)	
Yöntemi ile Algoritma Tasarımı	75
Özet	79

Kendimizi Sınavalım	80
Kendimizi Sınavalım Yanıt Anahtarı	81
Sıra Sizde Yanıt Anahtarı	81
Yararlanılan ve Başvurulabilecek Kaynaklar	82

Algoritma Analizi..... 84

5. ÜNİTE

GİRİŞ	85
ALGORİTMA ANALİZİ İÇİN MATEMATİKSEL ARKA PLAN	86
FONKSİYONLARIN BüYÜMESİ	87
ALGORİTMALARIN EN KÖTÜ DURUM, EN İYİ DURUM VE ORTALAMA DURUM VERİMLİLİKLERİ	87
ASİMPİTİK GÖSTERİMLER	88
Büyük O Gösterimi	89
Büyük Ω Gösterimi	89
Büyük Θ Gösterimi	90
ÖZYİNELEMELİ OLMAYAN ALGORİTMALARIN ANALİZİ	91
Algoritma Analizi ile İlgili Genel Kurallar	92
Algoritma Analiz Örnekleri	93
ÖZYİNELEMELİ ALGORİTMALARIN ANALİZİ	95
Özet	98
Kendimizi Sınavalım	99
Kendimizi Sınavalım Yanıt Anahtarı	100
Sıra Sizde Yanıt Anahtarı	100
Yararlanılan ve Başvurulabilecek Kaynaklar	100

Arama Algoritmaları..... 102

6. ÜNİTE

GİRİŞ	103
ARDIŞIK ARAMA	103
İKİLİ ARAMA	105
ARAMA ALGORİTMALARININ KARŞILAŞTIRILMASI	111
Özet	112
Kendimizi Sınavalım	113
Kendimizi Sınavalım Yanıt Anahtarı	114
Sıra Sizde Yanıt Anahtarı	115
Yararlanılan ve Başvurulabilecek Kaynaklar	118

Sıralama Algoritmaları..... 120

7. ÜNİTE

GİRİŞ	121
BALONCUK SIRALAMASI	121
SEÇMELİ SIRALAMA	124
ARAYA SOKARAK SIRALAMA	127
HIZLI SIRALAMA	130
BİRLEŞTİREREK SIRALAMA	133
YİĞİN SIRALAMASI	135
SIRALAMA ALGORİTMALARININ ÖZELLİKLERİ	139
Özet	140
Kendimizi Sınavalım	141
Kendimizi Sınavalım Yanıt Anahtarı	142
Sıra Sizde Yanıt Anahtarı	143
Yararlanılan ve Başvurulabilecek Kaynaklar	147

8. ÜNİTE

Çizge Algoritmaları	148
GİRİŞ	149
ÇİZGELERLE İLGİLİ TEMEL KAVRAMLAR	149
ENİNE ARAMA ALGORİTMASI	151
ÖNCE DERİNLİĞİNE ARAMA ALGORİTMASI	157
DİJKSTRA EN KISA YOL ALGORİTMASI	162
Özet	169
Kendimizi Sınayalım	170
Kendimizi Sınayalım Yanıt Anahtarı	173
Sıra Sizde Yanıt Anahtarı	173
Yararlanılan ve Başvurulabilecek Kaynaklar	175

Önsöz

Sevgili öğrenciler,

Algoritma sözcüğü Türk Dili Kurumu tarafından “iyi tanımlanmış kuralların ve işlemelerin adım adım uygulanmasıyla bir sorunun giderilmesi veya sonuca en hızlı biçimde ulaşılması işlemi” şeklinde tanımlanmıştır. Algoritmanın bilgisayar bilimlerindeki karşılığı ise “belirli bir işi yapmak veya bir problemi çözmek için adım adım tanımlanmış işlemler kümesi” şeklinde ifade edilebilir. Bir algoritmayı belirtmek için metinsel olarak düz bir ifade kullanabiliriz veya anlaşılabilirlik derecesi çok daha yüksek olan akış diyagramlarından faydalana biliriz. Algoritmanın bir bilgisayar tarafından gerçekleştirmesi söz konusu olduğunda ise devreye programlama ve programlama dili girer. Algoritmalar ve Programlama kitabı size algoritma tasarımını, algoritma analizini, algoritmaların kullandığı verilerin bilgisayar ortamında etkin olarak saklanması ve işlenmesine olanak tanıyan temel veri yapılarını öğretmeyi amaçlamaktadır.

Kitabımızın ilk ünitesinde algoritma kavramı, algoritmaların temel özellikleri, algoritmaların gösteriminde kullanılan yöntemler ve algoritmaların sınıflandırılması konularına değinilmiştir. Ayrıca, programlamada kullanılan veri yapıları konusunda ön bilgi verilmiştir. Temel veri yapıları arasında yer alan diziler, bağlı listeler, kuyruklar ve yiğinlar ikinci ünitede; ağaçlar, yiğin ağaçları ve özetleme tabloları ise üçüncü ünitede açıklanmıştır. Dördüncü ünitede, algoritma tasarımını ve algoritma tasarımında kullanılan tekniklere değinilmiştir. Kitabımızın beşinci ünitesinde algoritma analizi anlatılmış olup, gerekli matematiksel arka plan, fonksiyonların büyümesi, algoritmaların en kötü durum, en iyi durum ve ortalama durum verimlilikleri, asimptotik gösterimler, özyinelemeli ve özyinelemelisiz algoritmaların analizi konuları anlatılmıştır. Altıncı ünitede, programlamada sıkılıkla ihtiyaç duyulan arama işlemleri için kullanılan başlıca arama algoritmaları açıklanmıştır. Yedinci ünitede, sıralama işlemleri için faydalanan temel sıralama algoritmaları ele alınmıştır. Kitabımızın sekizinci ve son ünitesinde ise çizge algoritmaları anlatılmış olup, çizgelerle ilgili temel kavramlar, çizgeleri temel alan arama algoritmaları ve en kısa yol algoritmaları hakkında bilgi verilmiştir.

Kitabımızda, algoritmaların gerçekleştirmesi için C programlama dili temel alınmış ve programlama örnekleri bu doğrultuda hazırlanmıştır. Her bir ünitede sunulan bilgiler, açıklayıcı örneklerle desteklenmiş, konulara olan hakkını yetinizin artmasına yönelik “Sıra Sizde” çalışmaları sunulmuş, ünite sonunda öğrendiklerinizi sınamanıza imkân tanıyan “Kendimizi Sınayalım” sorularına yer verilmiştir.

Algoritmalar ve Programlama kitabı yardımcıyla sahip olacağınız bilgi birikimi ve tecrübenin yalnızca eğitiminiz esnasında değil, meslek hayatınızda da fayda sağlamasını ümit eder, hepинize başarılar dilerim.

Editör

Prof.Dr. Serkan GÜNAL

ALGORİTMALAR VE PROGRAMLAMA

1

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Programlamanın temel unsurlarından biri olan algoritma kavramını açıklayabilecek,
 - 🕒 Algoritmaların gösteriminde kullanılan farklı yöntemleri listeleyebilecek,
 - 🕒 Çeşitli algoritma sınıflarını tanımlayabilecek,
 - 🕒 Verileri etkin bir şekilde organize etmeye yarayan veri yapısı kavramını açıklayabilecek
 - bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- Algoritma
- Akış Şeması
- Sözde Kod
- Özyineleme
- Dinamik Programlama
- Veri Yapısı

İçindekiler

Algoritmalar ve Programlama

Algoritma Kavramı ve
Programlama Temelleri

- GİRİŞ
- ALGORİTMA NEDİR?
- ALGORİTMA GÖSTERİM YÖNTEMLERİ
- ALGORİTMALARIN SINIFLANDIRILMASI
- VERİ YAPILARI

Algoritma Kavramı ve Programlama Temelleri

GİRİŞ

Kitabımızın ilk ünitesi olan “Algoritma Kavramı ve Programlama Temelleri” ünitesinde algoritma kavramı, algoritma gösterim yöntemleri, algoritma sınıfları ve veri yapıları yer almaktadır.

Algoritma konusuna giriş, algoritmanın tanımı ve algoritmaların temel özellikleri “Algoritma Nedir?” başlığında anlatılmıştır. Bu başlıkta günlük hayattan algoritma örnekleri verilerek, algoritma kavramının zihinlerimizde daha açık bir şekilde belirlenmesi hedeflenmiştir.

Algoritmaların gösteriminde kullanılan çeşitli yöntemler bulunmaktadır. “Algoritma Gösterim Yöntemleri” başlığında bu yöntemlerden başlıcaları işlenmiştir. Gösterilen her bir yöntem için açıklayıcı birer örnek verilerek, algoritmaların gösterim yöntemleri pekiştirilmiştir.

Algoritmalar, ilgilendikleri problemler için uyguladıkları çözüm yöntemine göre sınıflandırılabilir. Bir programın verimli ve etkin çalışabilmesi için programın hedefine uygun algoritmalar kullanılmalıdır. “Algoritmaların Sınıflandırılması” başlığında başlıca algoritma türleri gösterilmiş ve bu türlerin hangi problemler üzerinde uygulanabileceği açıklanmıştır.

Ünitenin son bölümünde “Veri Yapıları” anlatılmıştır. Bu başlıkta, veri yapısı kavramına giriş ve veri yapısının tanımı yer almaktadır. Ayrıca bu bölümde, kitabımız genelinde işlenecek veri yapılarının listesi de verilmiştir.

ALGORİTMA NEDİR?

Algoritma, bir işin nasıl yapılacağını tarif eden adımlar kümesidir. Günlük hayatımızın büyük kısmında, farkında olmadan da olsa algoritmalar ile karşı karşıya geliriz. Bir yemeğin yapılmasındaki adımları içeren yemek tarifi, yerini bilmemişimiz bir restoranı bulmamıza yardımcı olan yol tarifi, bir elektronik cihazın kullanım kılavuzu, algoritmaların günlük hayatımızdaki kullanımına örnek olarak gösterilebilir.

Günlük yaşamımızda karşılaştığımız algoritma örneklerini detaylı ve açık bir şekilde tarif etmek mümkündür. Banka hesabımızdan nakit para temin etmemizi sağlayan ATM'den para çekme algoritmasını adım adım inceleyelim:

- Hesabın bulunduğu bankaya ait bir ATM'ye gidilir.
- ATM önungündeki bekleme kuyruğunu girilir.
- İşlem sırası gelene kadar kuyrukta beklenir.
- İşlem sırası geldiğinde, bankamatik kartı ATM'nin kart haznesine takılır.
- Bankamatik kartına ait şifre girilir ve “Giriş” tuşuna basılır.

- Para çekme menüsüne erişilir.
- Çekilecek nakit tutarı belirlenir ve “Devam” tuşuna basılır.
- ATM, bankamatik kartını kart haznesinden çıkartır.
- Bankamatik kartı ATM'den geri alınır.
- ATM, nakit parayı para haznesine doldurur.
- Nakit para ATM'den alınır.
- Para çekme işlemi tamamlanarak, işlem kuyruğundan çıkarılır.

ATM'den para çekme algoritmasının yukarıda gösterilen adımlarında, bir kişinin para çekmek için yapması gerekenler listelenmiştir. Bu örnek doğrultusunda, bir algoritmayı oluşturan temel bileşenlerin, yapılacak işe yönelik açıklama ve işin yapılmasında izlenecek adımlar olduğu söylenebilir. Açıklama kısmında işin tanımı yapılır ve işle ilgili detaylar bildirilir. Adımlar kısmında ise işin başlangıcından sonuna kadar takip edilecek işlemler belirtilir.

SIRA SİZDE



1

Çamaşır makinesi kullanarak çamaşırları yıkamak, günlük hayatı karşımıza çıkan bir algoritma örneğidir. Bu eylemi bir algoritma şeklinde ifade ediniz ve adımlarını belirtiniz.

Algoritmaların Temel Özellikleri

Algoritma kavramı, programlamadaki temel unsurlardan birisidir. Yazılım dünyasında geliştirilen programlar, bilgisayarın yapacağı işlemleri algoritmalar aracılığıyla tarif eder. Bilgisayar programları aracılığıyla çözülmek istenen bir problem için uygun bir algoritma geliştirilemiyorsa, o problemin program ile çözülmesi mümkün değildir.

Algoritmalar kendi aralarında sınıflandırılabilir ve karşılaştırılabilir. Aynı işlevi gören algoritmalar, farklı adımlara sahip olabilir. Programcılar, kendi ihtiyaçları doğrultusunda en uygun algoritmayı tasarlamak ve kodlamak durumundadırlar.

Bir algoritmanın sahip olması gereken temel özellikler aşağıda listelenmiştir:

- *Girdi ve Çıktı Bilgisi*: Algoritmada girdi ve çıktı bilgileri olmalıdır. Girdi bilgisi algoritmeye dışarıdan verilirken, çıktı bilgisi ise algoritma içerisinde üretilir. Bu bilgiler, algoritma için tanımlı veri kümesine ait olmalıdır.
- *Açıklık*: Algoritmayı oluşturan adımlar doğru ve kesin bir şekilde tanımlanmalıdır.
- *Doğruluk*: Farklı girdi bilgileri ile çalışabilen algoritmalar, her girdi için doğru bir çıktı üretmelidir.
- *Sonluluk*: Algoritmaların daima bir sonu olmalıdır. Girilen veri boyutundan bağımsız bir şekilde, algoritma adımları farklı bir aşamaya geçebilmeli veya sonlanmalıdır. Algoritma adımları gerçekleştirilirken, algoritma sonsuz döngüye girmemelidir.
- *Verimlilik*: Algoritmayı oluşturan adımlar, yapılan iş için kabul edilebilir bir süre içerisinde tamamlanmalıdır.
- *Genellik*: Bir algoritma, aynı türdeki problemlerin hepsine uygulanabilir olmalıdır.

DİKKAT



Bir algoritmanın verimliliği, o algoritmayı kullanan programın performansını doğrudan etkiler. Programların hızlı ve verimli çalışması için algoritma tasarımasına özen gösterilmelidir.

ALGORİTMA GÖSTERİM YÖNTEMLERİ

Algoritmaların tanımlanmasında ve gösteriminde kullanılan farklı yöntemler mevcuttur. Bu yöntemlerden başlıcaları konuşma dili ile gösterim, akış şeması ile gösterim ve sözde kod (pseudocode) ile gösterimdir.

Konuşma Dili

Bir algoritmanın açıklaması ve algoritmada yer alan adımlar, konuşma dili kuralları çerçevesinde ifade edilebilir. Bu gösterim yönteminde, algoritma açık ve kesin bir dille tanımlanır. Algoritmada yer alan adımlar liste halinde yazılır.

İki pozitif tamsayının ortak bölenlerinin en büyüğünü bulmak için kullanılan Öklid (Euclid) algoritmasının açıklaması ve adımları Şekil 1.1'de, konuşma dili kullanılarak gösterilmiştir.

Şekil 1.1

Açıklama: Elimizde iki adet pozitif tamsayı vardır. Bu iki sayının ortak bölenlerinin en büyüğü bulunacaktır.

*Euclid
Algoritmasının
Konuşma Dili ile
Gösterimi.*

Adımlar:

- Adım 1.** Sayılardan büyük olanı A, küçük olanı B olarak isimlendir.
- Adım 2.** A sayısını B sayısına böl, kalanı K olarak isimlendir.
- Adım 3.** K sayısı 0'dan farklı ise, A sayısına B'nin değerini ata, B sayısına da K'nın değerini ata ve Adım 2'ye geri dön. K sayısı 0 ise, ortak bölenlerin en büyüğü B'nin değeridir.

Örnek: Elimizdeki sayılar 8 ve 12 olsun.

Adım 1. $A = 12$, $B = 8$

Adım 2. $A \% B = 4$, $K = 4$

Adım 3. $K = 4$ olduğundan $K \neq 0$, $A = 8$, $B = 4$, Adım 2'ye dön

Adım 2. $A \% B = 2$, $K = 0$

Adım 3. $K == 0$ olduğundan ortak bölenlerin en büyüğü B'nin değeri, yani 4'tür.

Akış Şeması

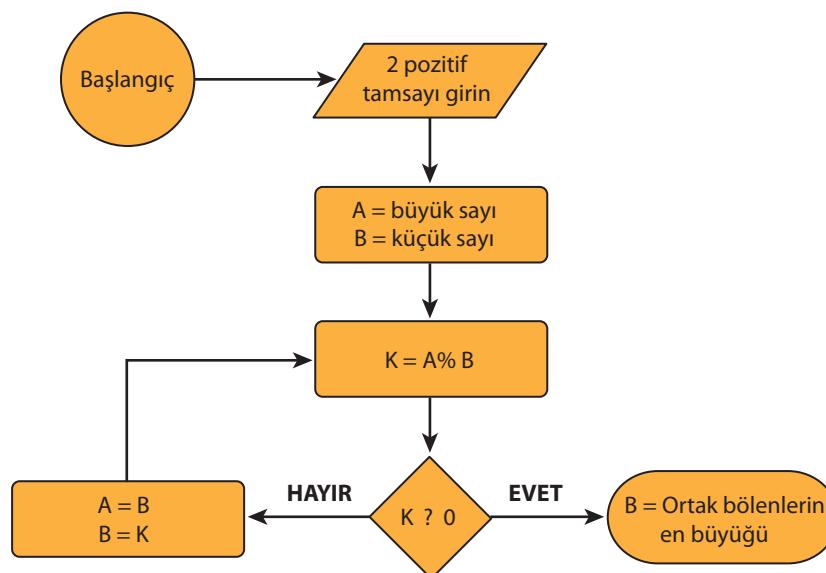
Akış şeması, algoritmaların gösteriminde kullanılan faydalı bir yöntemdir. Bir akış şemasında algoritma adımlarını ifade eden kutucuklar, adımlar arası geçişleri gösteren oklar, karar verme mekanizmaları olarak kullanılan şekiller bulunabilir.

Akış şeması, bir algoritmanın görsel halini ifade eder. Görsellik, algoritmaların daha kolay anlaşılabilmesine olanak sağlar. Programcılar ve çözümleyiciler tarafından yaygın olarak kullanılan akış şemalarını oluşturmak için birçok farklı çizim yazılımı bulunmaktadır.

İki pozitif tamsayının ortak bölenlerinin en büyüğünü bulmak için kullanılan Euclid algoritmasının akış şeması ile gösterimi Şekil 1.2'de verilmiştir.

Şekil 1.2

*Euclid Algoritmasının
Aks Şeması ile
Gösterimi.*



İNTERNET



Web tarayıcıları üzerinden çevrimiçi olarak kullanılabilen (ücretsiz veya deneme amaçlı) çizim yazılımlarından bazlarına www.draw.io, www.gliffy.com, www.lucidchart.com bağlantılarından ulaşabilirsiniz.

Sözde Kod

Sözde kod (pseudocode), bir algoritma veya program oluşturulurken kullanılan, konuşma diline benzer bir yapıya sahip, programlama dillerinin detaylarından uzak bir anlatım şeklidir.

Algoritmaların sözde kod ile gösterimi, oldukça yaygın ve etkili bir yöntemdir. Sözde kodlarda bir programlama diline benzeyen ifadeler kullanılsa da bu ifadelerin bilgisayar tarafından anlaşılması mümkün değildir.

Sözde kodlar, programlama mantığı ile konuşma dili cümlelerinin harmanlanmasıından meydana gelir ve herkes tarafından rahatlıkla anlaşılabilir. Sözde kodu okuyan bir kişi, programlama dillerinin detaylarına takılmadan, algoritmanın çalışma mantığını kavrayabilir.

İki pozitif tamsayının ortak bölenlerinin en büyüğünü bulmak için kullanılan Euclid algoritmasının sözde kodu Şekil 1.3'te gösterilmiş olup, adımları aşağıdaki gibi özetlenebilir.

- 1 numaralı adımda, algoritmanın tanımı yapılmış ve iki pozitif tamsayının girdi olarak kullanılacağı belirtilmiştir.
- 2 numaralı adımda, A sayısının B sayısına bölümünden kalan değer, K sayısına eşitlenmiştir.
- 3 ve 6 numaralı adımlar arasında basit bir döngü kurulmuştur. Bu döngü, K sayısı 0'a eşit olana kadar devam edecektir.
- 7 numaralı adımda döngünün sonlandığı belirtilmiştir.
- 8 numaralı adımda B sayısı ekrana yazdırılacaktır. Yazdırılan bu değer, algoritmanın girdileri için ortak bölenlerin en büyüğüdür.
- 9 numaralı adım, sözde koda ait son satırıdır. Bu adımda algoritmanın tamamlanlığı belirtilmiştir.

Şekil 1.3

```

1 procedure EUCLID (A, B : positive integers)
2     K = A mod B
3     while (K != 0)
4         A = B
5         B = K
6         K = A mod B
7     end while
8     print B
9 end procedure

```

*Euclid Algoritmasının
Sözde Kod ile
Gösterimi.*

ALGORİTMALARIN SINIFLANDIRILMASI

Algoritmalar, problemlerin çözümü için uyguladıkları yönteme göre sınıflandırılabilir. Algoritmaları sınıflandırmadaki temel amaç, problemlerin çözümünde başvurulabilecek değişik metotları ve alternatifleri tespit edebilmektir.

Ünitemizin bu bölümünde, aşağıda listelenen algoritma türleri incelenecaktır:

- Özyinelemeli Algoritmalar (Simple Recursive Algorithms)
- Geri İzlemeli Algoritmalar (Backtracking Algorithms)
- Böl ve Yönet Algoritmaları (Divide and Conquer Algorithms)
- Dinamik Programlama (Dynamic Programming)
- Açıgözlü Algoritmalar (Greedy Algorithms)
- Kaba Kuvvet Algoritmaları (Brute Force Algorithms)

Özyinelemeli Algoritmalar (Simple Recursive Algorithms)

Kendisini doğrudan veya dolaylı olarak çağrıran algoritmalar özyinelemeli algoritma adı verilir. Bu algoritmalarla, problemler daha küçük ve basit parçalara indirgenir. Küçük parçalar için oluşturulan çözümlerin birleştirilmesiyle ana problemin çözümü elde edilir.

Faktöriyel hesabı, özyinelemeli bir algoritma kullanılarak çözülebilecek problemlere güzel bir örnektir. 5 sayısının faktöriyeli bulunmak istendiğinde, 5'ten 1'e kadar olan tam-sayılar çarpılır. Bu problemin özyinelemeli bir algoritma ile çözümünde aşağıdaki adımlar uygulanır:

- Algoritmanın çıkış koşulu belirlenir ($1! = 1$).
- 2 sayısının faktöriyeli hesaplanır ($2! = 1! * 2 = 2$).
- 3 sayısının faktöriyeli hesaplanır ($3! = 2! * 3 = 6$).
- 4 sayısının faktöriyeli hesaplanır ($4! = 3! * 4 = 24$).
- 5 sayısının faktöriyeli hesaplanır ($5! = 4! * 5 = 120$).
- Beklenen hesaplama ulaşıldığı için algoritma sonlandırılır.

Faktöriyel hesabını özyinelemeli bir algoritma aracılığıyla bulan C fonksiyonu, Örnek 1.1'de gösterilmiştir.

ÖRNEK 1.1

Faktöriyel hesabını özyinelemeli şekilde yapan fonksiyon.

```
int factorial(int n) {
    if(n <= 1) {
        return 1;
    }
    return n * (factorial(n-1));
}
```

Geri İzlemeli Algoritmalar (Backtracking Algorithms)

Geri izlemeli algoritmalar, genellikle optimizasyon problemlerinde kullanılan, problem çözümünde tüm olasılıkları deneyen algoritmalarıdır. Bu algoritmalarda çözüm kademeli şekilde oluşturulur. Algoritma çözüm aşamasında ilerlerken, olası çözüm yollarının hepsini deneyerek bir sonraki adıma geçmeye çalışır. Algoritmanın denediği çözüm yolundan sonuç alınamazsa, algoritma bir önceki adımda bulunan diğer olası çözüm yollarına geri döner.

Geri izlemeli algoritmaların kullanımı ile çözülen birçok problem vardır. Bu problemlerin başlıcaları aşağıda listelenmiştir:

- Sudoku: 9 x 9'luk bir tablonun her satır ve sütununda 1'den 9'a kadar sayıların olması gerekmektedir. Bazı değerlerin dolu olarak verildiği bulmaca nasıl çözülür?
- Sekiz Vezir Problemi: Sekiz vezir, bir satranç tahtasına birbirlerine hamle yapamayacak şekilde nasıl yerleştirilir?
- Sırt Çantası Problemi: Elimizde kapasitesi belirli bir sırt çantası, ağırlığı ve değeri belirli nesneler vardır. Sırt çantasına hangi nesneler doldurulduğunda, çantaya konan nesnelerin toplam değeri en fazla olur?

SIRA SİZDE



Geri izlemeli algoritmalar ile çözülebilcek sırt çantası problemini ele aldığımızda, problemi çözecek algoritmayı geliştirirken ne gibi önkoşulları dikkate almamız gereklidir?

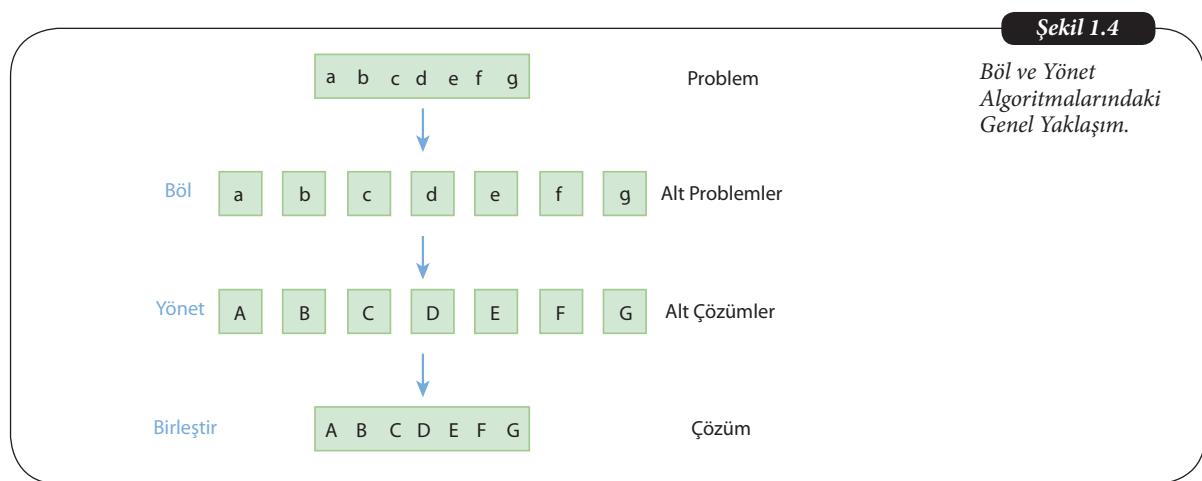
Böl ve Yönet Algoritmaları (Divide and Conquer Algorithms)

Böl ve yönet algoritmaları, problemlerin mümkün olan en küçük alt parçalara ayrıldığı, her bir alt parçanın diğerlerinden bağımsız şekilde çözüldüğü algoritmalarıdır. Problemin genel çözümü elde edilirken alt parçalara ait çözümler belirli bir sırayla bir araya getirilir.

Böl ve yönet algoritmaları, genellikle üç ana aşamadan meydana gelmektedir:

- Bölme (Divide): Problemin daha küçük parçalara ayrıldığı aşamadır. Problem daha alt parçalara bölünmeyecek hale gelene kadar, özyinelemeli bir yaklaşımla bölme işlemi gerçekleştirilir.
- Yönetme (Conquer): Problemin alt parçalarının, birbirlerinden bağımsız olarak çözüldüğü aşamadır.
- Birleştirme (Merge): Problemin alt parçalarına ait çözümlerin, özyinelemeli bir yaklaşımla birleştirildiği aşamadır.

Böl ve yönet algoritmalarındaki genel yaklaşım Şekil 1.4'te görsel olarak açıklanmıştır.



Dinamik Programlama (Dynamic Programming)

Dinamik programlama, karmaşık problemleri küçük parçalar halinde çözen, elde edilen sonuçları bilgisayar hafızasında bir veri yapısında saklayan, genel çözümü elde ederken de veri yapılarında saklanan sonuçları kullanan bir programlama yöntemidir.

Bir problemin dinamik programlama ile çözülebilmesi için problemin alt parçalara ayrılabilmesi ve genel çözümün bu alt parçalardan oluşturulabilmesi gerekmektedir. Dinamik programlama yaygın olarak optimizasyon problemlerinde kullanılır.

Daha önce özyinelemeli algoritma aracılığıyla çözduğumuz faktöriyel hesabını dinamik programlama ile de çözebiliriz. Faktöriyel hesabını dinamik programlama ile yapan C programı Örnek 1.2'de gösterilmiştir.

Ezberleme (Memorization):
Bir problemin alt kümelerinin çözümlerini tekrar hesaplamak yerine, bilgisayar hafızasında saklayan yöntemdir.

Faktöriyel hesabını dinamik programlama ile yapan program.

ÖRNEK 1.2

```
#include<stdio.h>

int memory[100] = {1, 1};

int factorial(int n) {
    if(n <= 1) {
        return 1;
    }
    else {
        int i;

        for(i=2; i<=n; i++) {
            if(memory[i] == 0) {
                memory[i] = i * memory[i-1];
            }
        }
        return memory[n];
    }
}

int main(void) {
    int n;

    for(n=1; n<10; n++) {
        printf("%d! = %d\n", n, factorial(n));
    }

    getch();
    return 0;
}
```

Açgözlü Algoritmalar (Greedy Algorithms)

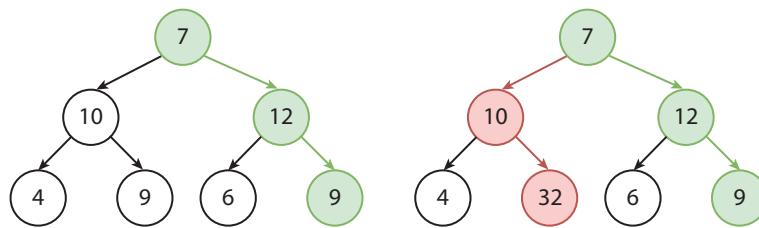
Bir problem için mümkün olan en doğru çözümü hedefleyen algoritmalar açgözlü algoritmalar adı verilir. Açgözlü algoritmalarla yerel olarak optimum sonuç elde edilirken, bulunan sonuç her zaman için en iyi çözüme karşılık gelmeyebilir.

Açgözlü algoritmalar ile problem çözümündeki temel yaklaşım, problemin küçük bir alt kümesi için çözüm oluşturmak ve bu çözümü problemin geneline yaymaktır. Algoritma içerisinde yapılan bir seçim, o an için doğru olsa bile sonraki seçimlerde olumsuz etki yapabilir.

Bir şehirden yola çıkan gezginin en fazla seyahat edeceği yolu hesaplama problemi, açgözlü bir algoritma ile çözülebilir. Bu yöntemde, mevcut şehirden gidilebilecek en uzak şehrde gidilir. Şekil 1.5'teki ilk haritada gezgin $7 + 12 + 9 = 28$ ile en uzak mesafeyi kat eder ve problemin en iyi çözümünü elde eder. İkinci haritada ise; gezgin yine $7 + 12 + 9 = 28$ yolunu kullanır fakat en iyi çözümü elde edemez. Açgözlü algoritmanın yerel optimumda bulamadığı $7 + 10 + 32 = 49$ yolu, en fazla seyahat edilen yol olmalıdır.

Sekil 1.5

En Fazla Yol Kat
Etme Probleminin
Açgözlü Algoritma ile
Çözümü.



a. Açgözlü algoritma ile en iyi çözüm b. Açgözlü algoritma ile optimum çözüm (en iyi çözüm değil)

Kaba Kuvvet Algoritmaları (Brute Force Algorithms)

Bir problemin çözümü aşamasında, kabul edilebilir bir çözüm elde edene kadar tüm olasılıkları deneyen algoritmalar kaba kuvvet algoritmaları denir.

Kaba kuvvet algoritmaları, genellikle problemin tanımından yola çıkarak en basit çözüm yolunu uygular ve rahatlıkla kodlanır. Fakat bu algoritmalarla çok fazla işlem yapılır ve çözüm yolu optimumdan uzaktır. Problemdeki veri hacmi büyükçe, kaba kuvvet algoritması ile çözüm şansı da azalır.

Bir liste içerisinde eleman aramak, kaba kuvvet algoritmaların kullanımıyla çözülebilecek problemlere bir örnektir. Listenin tüm elemanları sırayla kontrol edilerek, aranan elemanın listede olup olmadığına bakılabilir. Listenin eleman sayısı arttıkça, kaba kuvvet algoritmasının çalışma süresi ve yaptığı karşılaştırmalar da artacaktır.

SIRA SİZDE

3

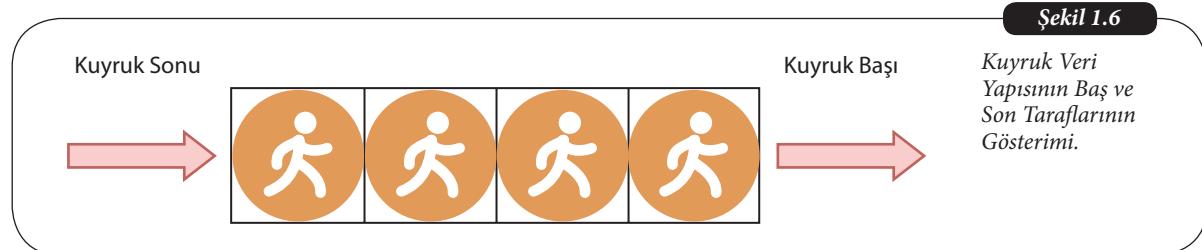
Bir tamsayı dizisinde belirli bir tamsayıyı arayacak kaba kuvvet algoritması kodlayınız. Geçitlereceğiniz algoritma, aranan eleman dizide bulunursa 1 değerini, bulunamazsa 0 değerini çıktı olarak vermelidir.

VERİ YAPILARI

Üst düzey programlamada veri içerisinde arama yapmak, veriye hızlı bir şekilde ulaşmak, bilgisayarın işlemcisini verimli kullanmak, aynı anda birçok isteğe cevap verebilmek gibi gereksinimler söz konusudur. Bilgisayar programlarının karmaşıklığı ve programda işlenen veri büyüklüğü arttıkça, verilerin daha sistematik ve verimli yönetilmesi gerekir.

Bilgisayar programlarında verilerin sistematik ve etkili bir şekilde organize edilmesi için veri yapıları kullanılır. Bir veri yapısı, içeriği elemanların mantıksal düzeni ve elemanlar üzerinde yapılabilecek işlemler ile tanımlanır.

Veri yapılarına örnek olarak, oldukça sık kullanılan bir veri yapısı çeşidi olan kuyruk Şekil 1.6'da gösterilmiştir. Her kuyruğun bir başı ve sonu olur. Veriler kuyruğun baş tarafından girerken, son tarafından çıkar. Bu özellikten dolayı kuyruk veri yapısını bir marketteki ödeme sırasına benzetebiliriz.



Kitabımız genelinde ayrıntıyla inclenecek ve örneklenenek veri yapıları şunlardır:

- Diziler (Arrays)
- Bağlı Listeler (Linked Lists)
- Kuyruklar (Queues)
- Yiğinlar (Stacks)
- Ağaçlar (Trees)
- Yiğin Ağaçları (Heaps)
- Öztleme Tabloları (Hash Tables)
- Çizgeler (Graphs)

Özet



Programmanın temel unsurlarından biri olan algoritma kavramını açıklamak

Bir işin nasıl yapılacağını tarif eden adımlar kümesine algoritma denir. Bir algoritmayı oluşturan temel bileşenler yapılacak işe yönelik açıklama ve işin yapılmasıyla izlenecek adımlardır. Algoritmanın açıklama kısmında işin tanımı yapılır ve işle ilgili detaylar belirlenir. Algoritmanın adımları kısmında ise işin başlangıcından sonuna kadar takip edilecek işlemler belirtilir.

Bir algoritmanın sahip olması gereken temel özellikler girdi ve çıktı bilgisine sahip olmak, açıklık, doğruluk, sonluluk, verimlilik ve genelliktir. Bu özelliklerden herhangi birinin eksik veya yetersiz olması, algoritmanın problem çözümü üzerindeki etkinliğini olumsuz etkiler.



Algoritmaların gösteriminde kullanılan farklı yöntemleri listelemek

Algoritmaların tanımlanmasında ve gösteriminde kullanılan başlıca yöntemler konuşma dili, akış şeması ve sözde kod olarak listelenebilir.

Algoritmaların konuşma dili ile gösteriminde algoritmanın açıklaması ve adımları konuşma dili kuralları çerçevesinde anlatılır. Algoritma net bir dille tanımlanır ve adımlar liste halinde yazılır.

Akış şeması, algoritmaların görsel halini sunan, okunurlüğünü ve anlaşılırlığını artıran faydalı bir gösterim yöntemidir. Akış şemalarında algoritma adımlarını ifade eden kutucuklar, adımlar arası geçişleri gösteren oklar, karar verme mekanizmaları olarak kullanılan şekiller bulunabilir.

Sözde kod (pseudocode), bir algoritma veya program oluştururken kullanılan, konuşma diline benzer bir yapıya sahip, programlama dillerinin detaylarından uzak anlatımlardır. Algoritmaların sözde kod ile gösterimi, programlama mantığı ile konuşma dili cümlelerini birleştirir. Bu sayede sözde kodu inceleyen kişi, algoritma mantığını rahatlıkla kavrar.



Çeşitli algoritma sınıflarını tanımlamak

Algoritmaları problemler için uyguladıkları çözüm yöntemlerine göre sınıflandırmak mümkündür. Ünite kapsamında Özyinelemeli Algoritmalar, Geri İzlemeли Algoritmalar, Böl ve Yönet Algoritmaları, Dinamik Programlama, Açıglı Algoritmalar ve Kaba Kuvvet Algoritmaları incelenmiştir. Özyinelemeli Algoritma-

lar, kendisini doğrudan veya dolaylı olarak çağırılan algoritmalarıdır. Problemler daha küçük parçalara indirgenir ve bu parçalar için oluşturulan çözümler birleştirilerek ana problemin çözümü elde edilir. Geri İzlemeли Algoritmalar, problemin çözüm aşamasında ilerlerken, olası çözüm yollarının hepsini deneyerek bir sonraki adıma geçmeye çalışan algoritmalardır. Denenen çözüm yolundan sonuc alınamazsa, algoritma bir önceki adımda bulunan diğer olası çözüm yollarına döner. Böl ve Yönet Algoritmaları, problemlerin mümkün olan en küçük alt parçalara ayrıldığı, her bir alt parçanın diğerlerinden bağımsız şekilde çözüldüğü algoritmalardır. Dinamik Programlama, karmaşık problemleri küçük parçalar halinde çözen, elde edilen sonuçları bilgisayar hafızasında bir veri yapısında saklayan, genel çözümü elde ederken de veri yapılarında saklanan sonuçları kullanan bir programlama yöntemidir. Açıglı Algoritmalar, bir problem için mümkün olan en doğru çözümü hedefleyen algoritmalarıdır. Kaba Kuvvet Algoritmaları, bir problemin çözüm aşamasında, yeterli bir çözüm elde edene kadar tüm olasılıkları deneyen algoritmalarıdır.



Verileri etkin bir şekilde organize etmeye yarayan veri yapısı kavramını açıklamak

Bilgisayar programlarında verilerin sistematik ve etkili bir şekilde organize edilmesi için veri yapıları kullanılır. Bir veri yapısı, içeriği elemanların mantıksal düzeni ve elemanlar üzerinde yapılabilecek işlemler ile tanımlanır.

Kitabımız genelinde incelenecek veri yapıları diziler (arrays), bağlı listeler (linked lists), kuyruklar (queues), yiğinlar (stacks), ağaçlar (trees), yiğin ağaçları (heaps), özetleme tabloları (hash tables) ve çizgelerdir (graphs).

Kendimizi Sınayalım

1. Aşağıdakilerden hangisi bir algoritmada beklenen özelliklerden biri **değildir**?

- a. Aynı türdeki problemlerin hepsi için geçerli olmak
- b. Doğru ve kesin adımlara sahip olmak
- c. Sonsuz döngüye girmek
- d. Dışarıdan girdi bilgisi alabilmek
- e. Farklı girdi bilgileri ile çalışabilmek

2. Bir algoritmayı oluşturan adımların doğru ve kesin bir şekilde tanımlanması, o algoritmanın hangi özelliğini temsil eder?

- a. Verimlilik
- b. Açıklik
- c. Sonluluk
- d. Genellik
- e. Doğruluk

3. Aynı türdeki iki problemden yalnız birini çözebilen bir algoritma, hangi temel algoritma özelliğini **karşılayamamaktadır**?

- a. Sonluluk
- b. Girdi ve çıktı bilgisi
- c. Açıklik
- d. Genellik
- e. Verimlilik

4. Bir algoritmayı görselleştirirken kutucukların, geçiş oklarının ve karar verme mekanizmalarının kullanıldığı gösterim yöntemi hangisidir?

- a. Konuşma dili
- b. Sözde kod
- c. Kaynak kodu
- d. Akış şeması
- e. Derleyici

5. Sözde kod ile temsil edilen aşağıdaki algoritmanın amacı nedir?

```

1 procedure SWAP(a, b : integers)
2     temp = a
3     a = b
4     b = temp
5 end procedure

```

- a. İki tamsayıdan büyük olanı bulmak
- b. Pozitif doğal sayılarla toplama işlemi yapmak
- c. Bir tamsayı kümесinin eleman sayısını hesaplamak
- d. Bir tamsayı kümесinin en küçük elemanını bulmak
- e. İki tamsayıın değerlerini kendi aralarında değiştirmek

6.

```

int fibonacci (int n) {
    if (n <= 0) {
        return 0;
    }
    else if (n == 1) {
        return 1;
    }
    else {
        return fibonacci (n-1) + fibonacci (n - 2);
    }
}

```

Yukarıdaki kod bloğunda tanımlanan fonksiyon, hangi algoritma sınıfına aittir?

- a. Özyinelemeli algoritmalar
- b. Kaba kuvvet algoritmaları
- c. Açıgözlü algoritmalar
- d. Böl ve yönet algoritmaları
- e. Geri izlemeli algoritmalar
- 7.** Bir çelik kasanın kilidini açmak için olası tüm şifreleri teker teker deneyen kişinin kullandığı yöntem, hangi algoritma sınıfına girer?
 - a. Kaba kuvvet algoritmaları
 - b. Açıgözlü algoritmalar
 - c. Geri izlemeli algoritmalar
 - d. Böl ve yönet algoritmaları
 - e. Dinamik programlama
- 8.** Dinamik programlamada, bir problemin alt kümelerinin çözümlerini tekrar tekrar hesaplamak yerine bilgisayar hafızasında saklayan yönteme ne ad verilir?
 - a. Programlama
 - b. Sorgulama
 - c. Denetleme
 - d. Hesaplama
 - e. Ezberleme
- 9.** Bilgisayar programlarında verilerin sistematik ve organizel bir şekilde saklanması sağlayan kavram hangisidir?
 - a. Programlama dili
 - b. Derleyici
 - c. Veri yapısı
 - d. Ağ bağlantısı
 - e. Monitör
- 10.** Aşağıdakilerden hangisi bir veri yapısı türü **değildir**?
 - a. Yığın
 - b. Gösterici
 - c. Kuyruk
 - d. Bağlı liste
 - e. Ağaç

Kendimizi Sınayalım Yanıt Anahtarı

1. c Yanınız yanlış ise “Algoritma Nedir?” konusunu yeniden gözden geçiriniz.
2. b Yanınız yanlış ise “Algoritma Nedir?” konusunu yeniden gözden geçiriniz.
3. d Yanınız yanlış ise “Algoritma Nedir?” konusunu yeniden gözden geçiriniz.
4. d Yanınız yanlış ise “Algoritma Gösterim Yöntemleri” konusunu yeniden gözden geçiriniz.
5. e Yanınız yanlış ise “Algoritma Gösterim Yöntemleri” konusunu yeniden gözden geçiriniz.
6. a Yanınız yanlış ise “Algoritmaların Sınıflandırılması” konusunu yeniden gözden geçiriniz.
7. a Yanınız yanlış ise “Algoritmaların Sınıflandırılması” konusunu yeniden gözden geçiriniz.
8. e Yanınız yanlış ise “Algoritmaların Sınıflandırılması” konusunu yeniden gözden geçiriniz.
9. c Yanınız yanlış ise “Veri Yapıları” konusunu yeniden gözden geçiriniz.
10. b Yanınız yanlış ise “Veri Yapıları” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

Programmanın temel kavramlarından biri olan algoritmalar, günlük hayatımızda da sıklıkla karşımıza çıkmaktadır. Çamaşır makinesi ile çamaşır yıkama eylemi de bir algoritma yardımıyla tarif edilebilir. Çamaşır makinesi ile çamaşır yıkama algoritması, kirli çamaşırların yıkanmasını ve temizlenmesini tarif eder. Bu algoritmanın olası adımları aşağıda belirtilmiştir:

- Çamaşır makinesinin kapağı açılır.
- Yıkanaçlı çamaşır makineye doldurulur.
- Çamaşır makinesinin kapağı kapatılır.
- Çamaşır uygundan temizlik maddeleri (deterjan, yumuşatıcı, leke çıkarıcı, vs.), deterjan haznesine doldurulur.
- Uygun yıkama ayarları (yıkama programı, sıcaklık, sıkma, vs.) yapılır.
- Makinenin elektrik bağlantısının sağlandığından emin olunur.
- Makinenin yıkama düğmesine basılır.
- Yıkama işlemi bittikten sonra çamaşır makineden boşaltılır.

Sıra Sizde 2

Sırt çantası problemindeki temel amaç, çanta içeresine doldurulan eşyalar ile en fazla değeri elde edebilmektir. Bu op-

timizasyon probleminin çözümü için uygulanacak yöntemi “yükte hafif, pahada ağır” sözü ile ilişkilendirebiliriz. Problemin çözümü için öncelikle çantanın taşıyabileceği en fazla ağırlık (çantanın kapasitesi) bilinmelidir. Çözüm için bilinmesi gereken bir diğer nokta ise her bir eşyanın ağırlığı ve maddi değeridir. Ayrıca, hiçbir eşyanın ağırlığı ve maddi değeri negatif olmamalıdır.

Sıra Sizde 3

Aşağıda verilen program, bir tamsayı dizisinde belirli bir sayıya aramak için geliştirilmiştir. Programda yer alan “search” fonksiyonu, dizinin tüm elemanlarını kontrol eden kaba kuvvet bir algoritma içermektedir. Dizinin elemanları sırayla dolaşılır, herhangi bir eleman ile aranan değer eşit olduğunda algoritma 1 döndürerek sona ermektedir ve sonraki elemanların kontrol edilmesine gerek kalmamaktadır. Aranan değer, dizinin hiçbir elemanın değeri ile aynı değilse, algoritma 0 değeri döndürerek sona ermektedir.

```
#include<stdio.h>
#define N 10

int tamsayılar[N] = {1, 6, 9, 88, -5, 42, -73,
99, 3, 5};

int search(int a) {
    int i;

    for(i=0; i<N; i++) {
        if(a == tamsayılar[i]) {
            return 1;
        }
    }

    return 0;
}

int main(void) {
    int n;

    for(n=1; n<50; n++) {
        if(search(n) == 1) {
            printf("%d dizide vardır.\n", n);
        }
        else {
            printf("%d dizide yoktur.\n", n);
        }
    }

    getch();
    return 0;
}
```

Yararlanılan ve Başvurulabilecek Kaynaklar

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

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

Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

2

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Tek boyutlu dizi tanımlamayı ve dizinin elemanlarına değer atamayı ifade ede-bilecek,
 - 🕒 Çok boyutlu dizi yapılarını açıklayabilecek,
 - 🕒 Bağlı liste yapısını tanımlayabilecek ve bağlı liste çeşitlerini sıralayabilecek,
 - 🕒 Kuyruk yapısının mantığını açıklayacak ve temel kuyruk işlemlerini sıralaya-bilecek,
 - 🕒 Dizilerle ve bağlı listelerle kuyruk veri yapısını oluşturabilecek,
 - 🕒 Yiğin yapısının mantığını kavrayacak ve temel yiğin işlemlerini sıralayabilecek,
 - 🕒 Dizilerle ve bağlı listelerle yiğin veri yapısını oluşturabilecek
- bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- Veri Tipi
- Veri Yapısı
- Dizi
- İndis
- Bağlı Liste
- Gösterici
- Kuyruk
- Yiğin

İçindekiler

Algoritmalar ve Programlama

Diziler, Bağlı Listeler,
Kuyruklar ve Yiğinlar

- GİRİŞ
- DİZİLER
- BAĞLI LİSTELER
- KUYRUKLAR
- YİĞİNLER

Diziler, Bağlı Listeler, Kuyruklar ve Yığınlar

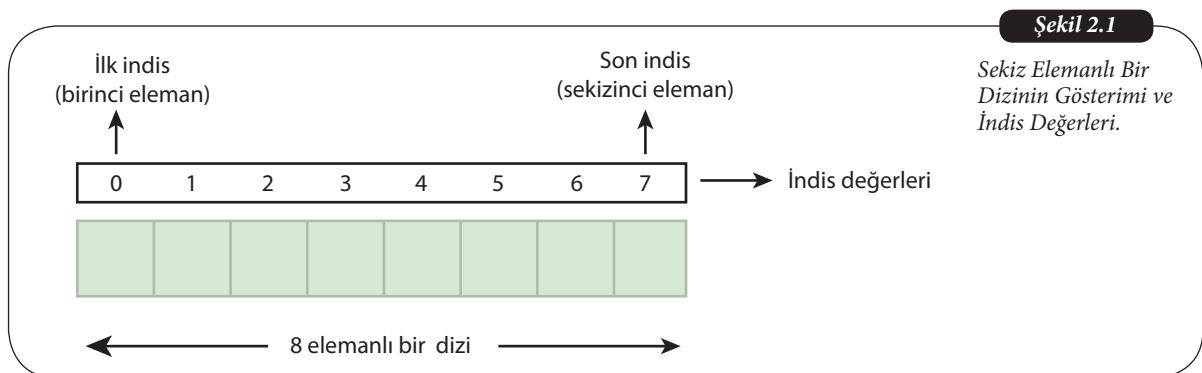
GİRİŞ

Kitabımızın bu ünitesinde işlenen konuların ana başlıklarını, programlamada temel veri yapılarından olan diziler, bağlı listeler, kuyruklar ve yığınlardır. Tek ve çok boyutlu dizi tanımlama, dizi elemanlarına değer atama, dizi elemanlarını gezinme ile ilgili temel bilgiler “Diziler” başlığında anlatılmıştır. Bağlı listelerin tanımı, bağlı liste türleri ve bağlı listelerdeki temel işlemler “Bağlı Listeler” başlığı altında verilmiştir. Kuyruk mantığı, kuyruk veri yapısının temel özellikleri, bu yapıda uygulanan işlemler ve kuyruk programlama yöntemleri “Kuyruklar” başlığında incelenmiştir. Yığın veri yapısının mantığı, bu yapıdaki temel işlemler ve yığın programlama yöntemleri ise “Yığınlar” başlığında anlatılmıştır.

Ünitenin genelinde işlenen veri yapıları, çeşitli örnekler ve Sıra Sizde çalışmaları ile desteklenmektedir. Örnek programların bilgisayar ortamında tekrar kodlanması, çalıştırılması, değiştirilmesi sizler için öğretici ve deneyim kazandırıcı olacaktır.

DİZİLER

Dizi (array), aynı tipteki verilerin tek bir değişken altında tutulmasını sağlayan veri yapısıdır. Sabit bir değere sahip olan dizinin uzunluğu, dizi oluşturulurken belirlenir. Bir dizide bulunan verilerin her biri, o dizinin bir elemanı olarak adlandırılır. Dizinin elemanlarına erişim indis (index) adı verilen sayısal değerler aracılığıyla sağlanır. İndislerin numaralandırılması 0 ile başlar, dizinin uzunluğunun 1 eksigine kadar ardışık olarak artarak devam eder. Örnek olarak, 8 elemanlı bir dizinin gösterimi ve indis değerleri Şekil 2.1'de yer almaktadır.



Diziler tek veya çok boyutlu olabilir. Ünite genelinde, çalışmalar tek boyutlu diziler üzerinden yapılacak, çok boyutlu dizilere ünite sonunda dezinilecektir.



DİKKAT

Dizilerin Tanımlanması

C dili ile programlamada bir dizi ile işlem yapabilmek için, diğer veri tiplerindeki değişkenlerde olduğu gibi, öncelikle dizinin tanımlanması gerekmektedir.

Sekil 2.2

Dizilerin
Tanımlanmasındaki
Genel İfade.

<dizi-tipi> <dizi-adı> [dizi-uzunluğu];

Dizilerin tanımlanmasındaki genel ifade
Şekil 2.2'de gösterilmiştir. Bu ifadeye göre:

- a. **dizi-tipi:** Dizinin hangi tipteki verilerden oluşacağını gösterir (int, char, double, float vb. veri tipleri olabilir).
- b. **dizi-adı:** Tanımlanan dizinin adını ifade eder.
- c. **dizi-uzunluğu:** Köşeli parantez içerisinde belirtilen bu değer, dizinin uzunluğunu belirtir.

ÖRNEK 2.1

int ve char tiplerinde tek boyutlu dizi tanımlamaları.

```
int tamsayıDizisi[10];
char karakterDizisi[20];
```

Örnek 2.1'deki kod satırlarında iki farklı dizi tanımlaması yapılmıştır. İlk dizi int veri tipinde, tamsayıDizisi adında, 10 eleman kapasiteli bir dizidir. İkinci dizi ise char veri tipinde, karakterDizisi adında, 20 eleman kapasiteli bir dizidir.

Dizilere Değer Atama

Bir diziyi veri tipi, isim ve kapasite belirterek tanımladığımızda (Örnek 2.1'de olduğu gibi), bilgisayar hafızasında dizi için bir yer ayrıılır; fakat dizi elemanlarına bir değer ataması yapılmaz.

Dizilere değer atmak için dizi tanımlaması ile birlikte veya tanımlamadan sonra kodlanan çeşitli komutlar vardır. Dizinin elemanları üzerinde gezinecek döngüler aracılığıyla dizi elemanları sıfırlanabilir, bir veri kaynağından (kullanıcı, metin dosyası, veri tabanı vb.) alınan değerler dizinin elemanlarına atanabilir, dizi elemanlarının değerleri programcı tarafından kod içerisinde belirtilebilir. Bu noktada önemli olan husus, programının ihtiyaci doğrultusunda en doğru yöntemin uygulanmasıdır.

ÖRNEK 2.2

Döngü kullanımı ile dizi elemanlarının sıfırlanması.

```
#include <stdio.h>

int main(void) {
    int dizi[6],
        i;

    for(i=0; i<6; i++) {
        dizi[i] = 0;
        printf("dizi[%d] = %d\n", i, dizi[i]);
    }

    getch();
    return 0;
}
```

Örnek 2.2'de ana fonksiyon içerisinde int veri tipinde, tamsayıDizisi adında, 6 eleman kapasiteli bir dizi tanımlanmıştır. Daha sonrasında bir döngü aracılığıyla dizinin her elemanına 0 değeri atanmıştır ve bu değerler ekrana yazdırılmıştır. Bu örnekteki programın ekran çıktısı, Program Çıktısı 2.1'de gösterilmiştir.

Program Çıktısı 2.1: Örnek 2.2'de gösterilen programın ekran çıktısı.

```
dizi[0] = 0
dizi[1] = 0
dizi[2] = 0
dizi[3] = 0
dizi[4] = 0
dizi[5] = 0
```

Tanımlama ile birlikte dizilere değer atama.

ÖRNEK 2.3

```
#include <stdio.h>
#define N 5

int main(void) {
    int dizi1[N] = {1, 2, 3, 4, 5};
    int dizi2[N] = {1, 2};
    int i;

    for(i=0; i<N; i++) {
        printf("dizi1[%d] = %d\t", i, dizi1[i]);
        printf("dizi2[%d] = %d\n", i, dizi2[i]);
    }

    getch();
    return 0;
}
```

Örnek 2.3'te ana fonksiyon içerisinde int veri tipinde, dizi1 ve dizi2 adlarıyla, 5 eleman kapasiteli iki dizi tanımlanmıştır. Birinci dizinin tanımlanmasında dizinin tüm elemanlarına değer atanmıştır. İkinci dizinin tanımlanmasında ise dizinin ilk iki elemanına değer atanmıştır. Bu dizinin diğer elemanları otomatik olarak 0 değerini almıştır. Daha sonra kurulan döngüde dizilerin elemanları sırayla ekrana yazdırılmıştır. Bu örnekteki programın ekran çıktısı, Program Çıktısı 2.2'de gösterilmiştir.

Program Çıktısı 2.2: Örnek 2.3'te gösterilen programın ekran çıktısı.

```
dizi1[0] = 1  dizi2[0] = 1
dizi1[1] = 2  dizi2[1] = 2
dizi1[2] = 3  dizi2[2] = 0
dizi1[3] = 4  dizi2[3] = 0
dizi1[4] = 5  dizi2[4] = 0
```

Bir dizinin elemanlarına değer atama işlemi, dizinin tanımlanmasıyla birlikte yapılırken dizinin boyutu belirtilmeyebilir. Örneğin, int dizi[] = {-3, 0, 3, 6}; şeklinde yapılan dizi tanımlamasında, dizinin boyutu derleyici tarafından algılanır ve hatasız çalıştırılır.

Veri girişi işlemini kullanıcıya yaptırmak mümkündür. C dilinde tanımlı scanf fonksiyonunu kullanarak, 5 elemanlı bir tamsayı dizisine veri girişi yapılan bir program yazınız.



SIRA SİZDE

1

Çok Boyutlu Diziler

Ünenin önceki kısımlarında incelediğimiz tek boyutlu dizilerin yanı sıra, programlama çok boyutlu diziler de oluşturulabilmektedir. Bu bölümde, çok boyutlu dizilerin en yaygın kullanılanları olan, iki boyutlu ve üç boyutlu diziler işlenecektir.

Iki Boyutlu Diziler

Satır ve sütunlardan oluşan tablolar şeklinde tanımlanabilen iki boyutlu diziler, çok boyutlu dizilerin en yalın halidir. m adet satır, n adet sütundan oluşan iki boyutlu bir dizi, toplam ($m \times n$) elemana sahip olabilir. 4 satır ve 3 sütundan oluşan iki boyutlu bir dizi, Şekil 2.3'te gösterilmiştir.

Şekil 2.3

Iki Boyutlu ($4 \times 3 = 4$ Satır, 3 Sütun) Bir Dizinin Gösterimi.

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]
a[2][0]	a[2][1]	a[2][2]
a[3][0]	a[3][1]	a[3][2]

İki boyutlu dizilerin elemanlarına değer atarken, her iki boyuttaki indisler üzerinde dolaşılmalıdır. Bu gereksinim için iç içe geçmiş 2 adet for döngüsü kurulabilir. İki boyutlu dizi tanımlama, bu dizinin elemanlarına değer atama ve atanmış değerleri ekrana yazdırma işlemlerini gerçekleştiren program, Örnek 2.4'te verilmiştir.

ÖRNEK 2.4

Iki boyutlu dizi tanımlama ve değer atama.

```
#include <stdio.h>
#define M 4
#define N 3

int main(void) {
    int dizi[M][N] = {{1,2,3}, {4,5,6}, {7}, 8,9,10};

    int i, j;

    for(i=0; i<M; i++) {
        for(j=0; j<N; j++) {
            printf("dizi[%d] [%d] = %d\n", i, j, dizi[i][j]);
        }
    }

    getch();
    return 0;
}
```

Örnek 2.4'te iki boyutlu bir tamsayı dizisi tanımlanmıştır. Bu dizinin birinci boyutunun uzunluğu 4, ikinci boyutunun uzunluğu ise 3 olarak atanmıştır. Dizi tanımlaması ile birlikte dizinin elemanlarına değer ataması da yapılmıştır. Dizinin elemanlarının değerleri, kurulan iç içe döngü sayesinde ekrana yazdırılmıştır. Bu programa ait ekran çıktısı ise Program Çıktısı 2.3'te gösterilmiştir.

Program Çıktısı 2.3: Örnek 2.4'te gösterilen programın ekran çıktısı.

```
dizi[0][0] = 1
dizi[0][1] = 2
dizi[0][2] = 3
dizi[1][0] = 4
dizi[1][1] = 5
dizi[1][2] = 6
dizi[2][0] = 7
dizi[2][1] = 0
dizi[2][2] = 0
dizi[3][0] = 8
dizi[3][1] = 9
dizi[3][2] = 10
```

Üç Boyutlu Diziler

Üç boyutlu diziler, iki boyutlu dizilerin katmanlar halinde bir araya gelmesiyle oluşur. Üç boyutlu bir diziyi, “*iki boyutlu dizilerin dizisi*” olarak da tanımlamak mümkündür. Boyut uzunlukları sırasıyla a, b, c olan üç boyutlu bir dizinin sahip olacağı toplam eleman sayısı $a * b * c$ kadar olur.

Üç boyutlu dizilerde dizi tanımlama ve elemanlara değer atama işlemleri, iki boyutlu dizilerdeki işlemlere benzerlik gösterir. Üç boyutlu dizilerin elemanları gezilirken, iç içe geçmiş 3 adet for döngüsü kurulabilir. Belirtilen işlemlerin yapıldığı bir program, Örnek 2.5'te verilmiştir.

Üç boyutlu dizi tanımlama ve değer atama.

ÖRNEK 2.5

```
#include <stdio.h>
int main(void) {
    int d[3][2][2] = {12,11,10,9,8,7,6,5,4,3,2,1};
    int i, j, k;
    for(i=0; i<3; i++) {
        for(j=0; j<2; j++) {
            for(k=0; k<2; k++) {
                printf("%d\n", d[i][j][k]);
            }
        }
    }
    getch();
    return 0;
}
```

Ünite genelinde verilen örnekleri bilgisayar ortamında denemeniz, diziler hakkında dene-yiminizin artmasına katkıda bulunacaktır.



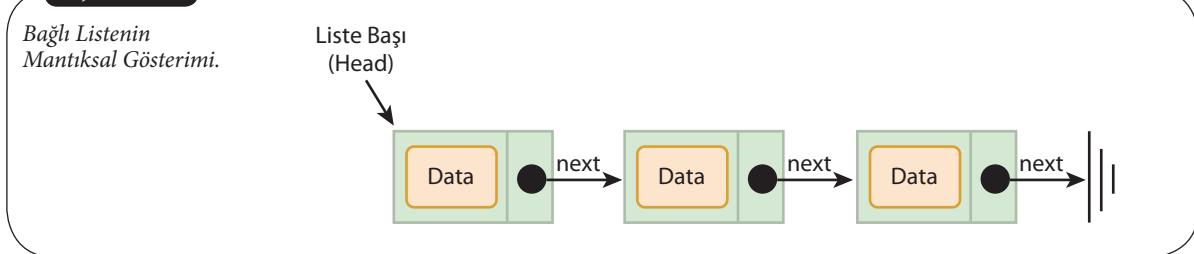
DİKKAT

BAĞLI LİSTELER

Bağlı liste (linked list), aynı türden nesnelerin doğrusal bir sırada ve birbirlerine bağlı şekilde saklandığı veri yapısıdır. Bağlı listedeki nesnelere düğüm (node) adı verilir ve düğümler birbirlerine bir sonraki düğümü işaret eden göstericiler (next pointer) aracılığıyla bağlanılmışlardır. Ayrıca, bağlı listelerde listenin başlangıcını işaret eden bir baş gösterici (head pointer) de bulunur.

Bağlı listeleri oluşturan düğümler genellikle iki kısımdan meydana gelir. Düğümün ilk kısmında veri saklanırken, ikinci kısmında ise bir sonraki düğümün bilgisayar hafızasındaki yeri saklanır (Şekil 2.4).

Şekil 2.4



Bağlı Listeler ile Dizilerin Karşılaştırması

Veri yapısı olarak benzerlik gösteren dizilerin ve bağlı listelerin, birbirleriyle kıyaslandığında, çeşitli açılarından avantajları ve dezavantajları bulunmaktadır. Bağlı listelerin ve dizilerin çeşitli ölcütlere göre karşılaştırması aşağıda verilmiştir:

- *Veri yapısı uzunluğu:* Dizilerde veri yapısının uzunluğu sabittir, gerekli durumlar da dizi uzunluğu artıramaz veya azıltılamaz. Bağlı listelerde uzunluk dinamiktir, yeni nesneler eklenebilir, var olan nesneler silinebilir.
- *Hafıza kullanımı:* Bağlı listelerdeki her bir nesnenin göstericisi için, bilgisayar hafızasında yer ayrılmazı gereklidir. Dizilerde böyle bir durum söz konusu değildir.
- *Veri ekleme/silme maliyeti:* Dizilerde ekleme ve çıkarma işlemleri, programlama açısından oldukça yüksek maliyetlidir. Bağlı listelerde ekleme veya çıkarma yapmak, dizilerdekine göre daha az maliyetli ve kolaydır.
- *Verilere doğrudan erişim:* Dizi elemanlarına indisler aracılığıyla doğrudan erişilebilir. Bağlı listelerde ise böyle bir durum söz konusu değildir. Bağlı listenin bir elemanına erişmek için o elemanın listede aranması ve bulunması gereklidir.

Bağlı Liste Türleri

Bağlı listelerin elemanları dolaşılırken ileriye doğru gitmek, geriye doğru hareket etmek ve listenin sonundan listenin başına erişmek mümkün olabilir. Belirtilen bu hareket kabiliyetleri, çeşitli türlerde bağlı listelerin ortayamasına neden olur. Bağlı listelerdeki üç tür aşağıda listelenmiştir:

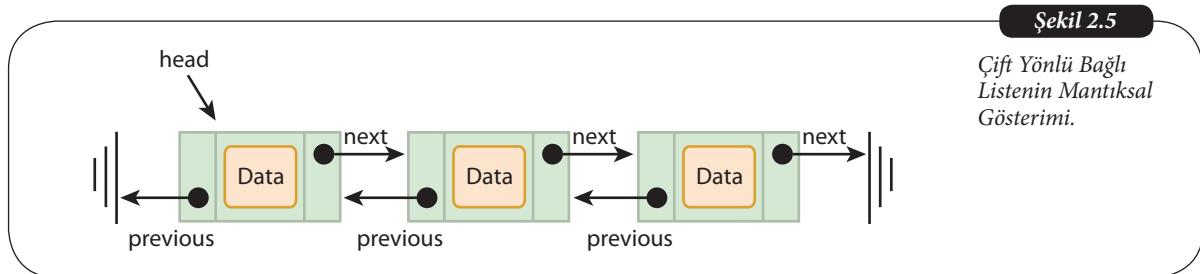
- i. Tek yönlü bağlı liste (Singly linked list)
- ii. Çift yönlü bağlı liste (Doubly linked list)
- iii. Dairesel bağlı liste (Circular linked list)

Tek Yönlü Bağlı Liste

Tek yönlü bağlı listelerde, liste düğümleri arasındaki gezinme yalnızca ileriye doğru gerçekleşir. Şekil 2.4'teki gösterim, tek yönlü bir bağlı liste örneğidir.

Çift Yönlü Bağlı Liste

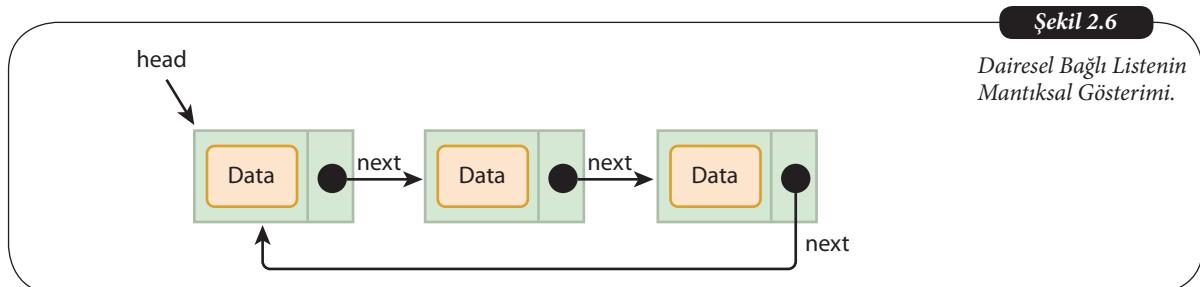
Çift yönlü bağlı listelerde, liste düğümleri arasında hem ileriye hem de geriye doğru gidelebilir. Çift yönlü bağlı listenin bir düğümü, bir sonraki düğümü işaret eden göstericisinin (next pointer) yanı sıra, bir önceki düğümü işaret eden göstericiyi (previous pointer) de içerir. Çift yönlü bağlı liste için bir örnek, Şekil 2.5'te gösterilmiştir.



Dairesel Bağlı Liste

Bir bağlı listenin son düğümünün bir sonraki düğümü işaret eden göstericisi (next pointer) listenin ilk düğümünü işaret ettiğinde liste dairesel hale gelmiş olur. Bağlı listelerin bu çeşidine dairesel bağlı liste denilmektedir. Dairesel bağlı liste için bir örnek, Şekil 2.6'da gösterilmiştir.

Gerek tek yönlü gerek çift yönlü bağlı listelerin dairesel hale getirilmesi mümkündür. Bazı kaynaklarda, dairesel bağlı listeler, kendi içerisinde tek yönlü veya çift yönlü olarak ikiye ayrılmaktadır.



Bağlı Listelerde Temel İşlemler

Bağlı listelerde listeye yeni eleman ekleme, listeden eleman çıkarma, listedeki toplam eleman sayısını bulma, listenin elemanlarını dolaşma, listede eleman arama, listenin ilk veya son elemanını bulma gibi çeşitli işlemler yapılabilir. Liste üzerinde yapılabilecek işlemler, listenin türüne göre de değişkenlik gösterir.

Bu bölümde, tek yönlü bağlı listelerdeki temel işlemler sıralanacak ve bu işlemler için örnek kodlar gösterilecektir.

Düğüm Yapısını Oluşturma

Bağlı listenin elemanlarını temsil etmek için bir düğüm yapısı oluşturulmalıdır. Bu yapıda düğümde saklanacak veri (data) ve bir sonraki düğümün göstericisi (next pointer) yer alır. Tamsayı değerler saklayacak bir bağlı liste için düğüm yapısı, Örnek 2.6'da verilmiştir.

ÖRNEK 2.6

Bağlı listede düğüm yapısı.

```
struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;
```

Listeye Eleman Ekleme (Insertion)

Bağlı listelerde listenin başına, sonuna veya herhangi bir bölgесine eleman eklemek mümkündür. Ekleme işleminde, eklenecek düğüm için malloc fonksiyonu ile hafızada yer açılır. Örnek 2.7'de, listenin başına ekleme yapan bir fonksiyon gösterilmiştir.

ÖRNEK 2.7

Bağlı listede liste başına eleman ekleme.

```
void insert(int a) {
    struct Node* t = (struct Node*) malloc(sizeof(struct Node));

    t->data = a;

    t->next = head;
    head = t;
}
```

Listeden Eleman Çıkarma (Deletion)

Bağlı listelerde listenin başından, sonundan veya herhangi bir bölgесinden eleman çıkarılabilir. Çıkarma işleminde çıkarılacak düğüm, free fonksiyonu ile hafızadan silinir. Çıkarma işlemi listenin ara bir bölgесinden yapılacaksa, öncelikle çıkarılacak düğüm liste içerisinde bulunmalıdır, düğüm bulunduktan sonra çıkışma işlemi gerçekleştirilmelidir. Bağlı listelerde çıkışma işlemini baş taraftan gerçekleştiren fonksiyon, Örnek 2.8'de yer almaktadır.

ÖRNEK 2.8

Bağlı listede liste başından eleman çıkışma.

```
void delete() {
    if(head != NULL) {
        struct Node *t = head;

        head = head->next;
        free(t);
    }
}
```

Listeyi Gezinme (Traversal)

Bağlı listenin elemanlarını gezinmek, listenin başından sonuna kadar gitmek demektir. Gezinme işlemi, listenin son elemanına ulaşılınca kadar, yani bir sonraki düğümü işaret eden göstericisi (next pointer) NULL olan eleman bulunana kadar devam eder. Listenin elemanları gezilirken, listede eleman arama veya elemanları ekrana yazdırma gibi işlemler de yapılabilir. Örnek 2.9'da listenin elemanlarını gezen kod bloğu gösterilmiştir.

Bağlı listenin elemanlarının gezilmesi.

ÖRNEK 2.9

```
void traverse() {
    struct Node *t = head;

    while(t != NULL) {
        printf("%d ", t->data);
        t = t->next;
    }
}
```

Tek bağlı listede temel işlemler.

ÖRNEK 2.10

```
#include<stdio.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void insertToHead(int a) {
    struct Node* t = (struct Node*) malloc(sizeof(struct Node));

    t->data = a;
    t->next = head;
    head = t;
}

void deleteFromHead() {
    if(head != NULL) {
        struct Node *t = head;
        head = head->next;
        free(t);
    }
}

void printList() {
    struct Node *t = head;

    while(t != NULL) {
        printf("%d ", t->data);
        t = t->next;
    }
}

int main(void) {
    int i, j;

    for(i=1; i<=10; i++) {
        insertToHead(i);
    }

    for(j=1; j<=4; j++) {
        deleteFromHead();
    }

    printList();
    getch();
}
```

Örnek 2.10'da temel işlemlerin uygulandığı bir tek yönlü liste programı gösterilmiştir. Bu programda liste başına eleman eklemek için `insertToHead()` fonksiyonu, liste başından eleman çıkarmak için `deleteFromHead()` fonksiyonu, listenin elemanlarını gezinerek ekrana yazdırmak için `printList()` fonksiyonu tanımlanmıştır.

Örneğin, ana fonksiyonunda (`main`) 1'den 10'a kadar çalışacak bir döngü oluşturulmuş ve döngünün her adımıda i değeri listenin başına eklenmiştir. Daha sonra, 1'den 4'e kadar çalışacak yeni bir döngü kurulmuş ve döngünün her adımıda listenin başındaki eleman silinmiştir. Sonuç olarak, listenin son hali üzerinde gezinme yapılmış ve listedeki elemanlar ekrana yazdırılmıştır. Bu örnek programa ait ekran çıktısı Program Çıktısı 2.4'te gösterilmiştir.

Program Çıktısı 2.4: Örnek 2.10'da gösterilen programın ekran çıktısı.

6	5	4	3	2	1
---	---	---	---	---	---

DİKKAT



Ünite genelinde verilen örnekleri okumanızın yanında bilgisayar ortamında bizzat denemeyiz, konuları özümsemeye katkı sağlayacaktır.

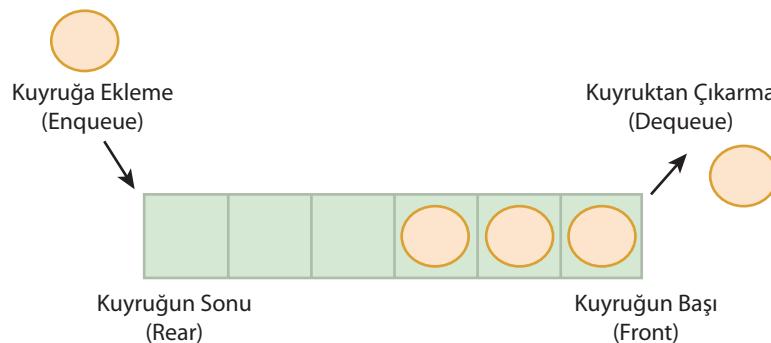
KUYRUKLAR

Günlük yaşamımızda kuyruğa girmek ve sıra beklemek oldukça rutin bir eylemdir. Bankalarda, gişelerde, süpermarketlerde, trafikte vb. birçok alanda insanlar sıralar oluşturur, işini bitiren kişiler kuyruktan ayrılır, yeni gelen kişiler kuyruğa dahil olur. Günlük hayatın taki bu kuyruk mantığı, programlamada da yer almaktadır.

Programlamada kuyruk (queue), verilerin doğrusal sırada tutulmasını sağlayan bir veri yapısıdır. Bir kuyruğun başı (front) ve sonu (rear) bulunur. Kuyruk yapısındaki temel işlemleri olan ekleme (enqueue) son taraftan, çıkışma (dequeue) ise baş taraftan gerçekleştirir. Dolayısıyla kuyruğa ilk giren eleman, kuyruktan ilk çıkan eleman olur (Şekil 2.7).

Şekil 2.7

Kuyruk Veri Yapısının
Mantıksal Gösterimi.



Kuyruk mantığının bir veri yapısı olarak programlanmasıında iki temel yöntem vardır:

- Dizilerin kullanımı ile kuyruk programlama
- Bağlı listelerin kullanımı ile kuyruk programlama

Dizi ile Kuyruk Uygulaması

Kuyruk veri yapısını programlarken bir diziden faydalabilir. Bu yöntemde, verileri tutacak bir diziye, kuyruğun başını takip edecek bir tamsayıya, kuyruğun sonunu takip edecek bir tamsayıya ve kuyruktaki mevcut eleman sayısını gösterecek bir tamsayıyı ih-

tiyâç duyulur. Kuyruk veri yapısının tamsayı tipinde değerleri saklayacağını varsayırsak, programda gerekli olan değişkenleri aşağıdaki gibi listeleyebiliriz:

- int queue[N]: Kuyruktaki elemanları tutacak N uzunluğunda tamsayı dizisi
- int front: Kuyruğun başını gösteren indis
- int rear: Kuyruğun sonunu gösteren indis
- int count: Kuyruktaki eleman sayısı

Kuyruk veri yapısının dizi ile uygulanması.

ÖRNEK 2.11

```
#include <stdio.h>
#define N 6

int queue[N];
int front = 0, rear = 0, count = 0;

void enqueue(int a) {
    if(count == N) {
        printf("Kuyruktaki yer yoktur.\n");
    }
    else {
        queue[rear] = a;
        rear++;
        if(rear == N) rear = 0;
        count++;
        printf("%d kuyruğa eklendi.\n", a);
    }
}

void dequeue() {
    if(count == 0) {
        printf("Kuyruktaki eleman yoktur.\n");
    }
    else {
        int a = queue[front];
        front++;
        if(front == N) front = 0;
        count--;
        printf("%d kuyruktan çıkarıldı.\n", a);
    }
}
```

Örnek 2.11'de, dizi ile kuyruk uygulamasının nasıl yapılabileceği gösterilmiştir. Kuyruktaki tamsayı verileri saklayacak diziyi queue adı verilmiş, dizinin boyutu ise 6 olarak tanımlanmıştır. Kuyruk yapısı için gerekli diğer değişkenler olan *front*, *rear* ve *count* tamsayıları programın başında 0 olarak atanmıştır. Kuyruğa ekleme yapmak için *enqueue()* fonksiyonu, kuyruktan çıkışma yapmak için *dequeue()* fonksiyonu tanımlanmıştır.

Ekleme işleminde öncelikle kuyruğun doluluğu (dizinin uzunluğu olan *N* ile *count* değişkenini karşılaştırarak) kontrol edilir ve kuyruktaki yer varsa işlemeye devam edilir. Kuyruğa eklenen bir sayı, *queue* dizisinin *rear* ile belirtilen indisine (kuyruğun sonunu) kaydedilir, *rear* değeri ve dizinin eleman sayısı 1 arttırılır.

Çıkarma işleminde öncelikle kuyruktaki eleman sayısı (*count* değişkeni ile 0 değerini karşılaştırarak) kontrol edilir ve kuyruktaki herhangi bir eleman varsa işlemeye devam edilir. Kuyruktan çıkarılacak bir değer, *queue* dizisinin *front* ile belirtilen indisinde yer almaktadır. Çıkarma işlemiyle *front* değeri 1 artırılırken, dizinin eleman sayısı 1 azaltılır. Örnekte, kuyruktan çıkarılan bir değer a sayısına atanarak ekrana yazdırılmıştır, fakat ekrana yazdırma dışında, bu değer kullanılarak farklı işlemler de gerçekleştirilebilir.

Ekleme ve çıkışma işlemlerinde önemli bir ayrıntı, *queue* dizisinde dairesel bir yapı kurulmasıdır. Eklemede *rear* değişkeni, çıkışmada ise *front* değişkeni dizinin eleman sayısına eşit olursa, ilgili değişken 0 değerine atanır. Böylelikle dizide dairesel bir yapı kurulur ve programın dizi indislerinden bağımsız, kesintisiz olarak çalışması sağlanır.

SIRA SİZDE

2

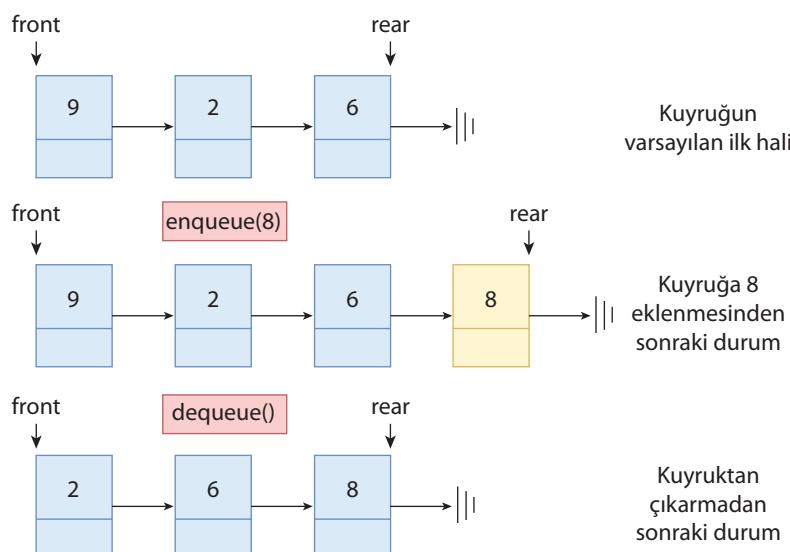
Örnek 2.11'i temel alarak, bir kuyruk uygulaması geliştirin. Yazacağınız programda kuyruğa sırasıyla 5, 12, 9, 8, 0, 1, 7 sayılarını ekleyin, çıkıştan çıkış işlemini 2 defa uygulayın, kuyruğa tekrar 5 sayısını ekleyin.

Bağılı Liste ile Kuyruk Uygulaması

Kuyruk yapısını programlamak için bir bağlı listeden de faydalabilir. Bu yöntemde bağlı listenin elemanlarını oluşturacak bir veri yapısına ve kuyruğun başını ve sonunu takip edecek göstereçilere ihtiyaç duyulur.

Şekil 2.8

Kuyruk Veri
Yapısında Bağlı Liste
Kullanımı.



Kuyruk veri yapısının bağlı liste ile gösterimi, Şekil 2.8'de belirtilmiştir. Kuyruğun ilk halinde 9, 2 ve 6 sayılarını içeren elemanlar bulunmaktadır. 9'yu içeren eleman kuyruğun başında, 6'yu içeren eleman ise kuyruğun sonundadır. Tanimlanan bu kuyruğa, enqueue işlemi ile 8'yi içeren bir eleman eklenmiştir ve kuyruğun sonu, bu elemanı gösterecek şekilde güncellenmiştir. Daha sonraki aşamada, dequeue ile kuyruğun başında bulunan kuyruktan çıkarılmıştır ve kuyruğun başı, 2'yi içeren elemanı göstererek şekilde güncellenmiştir.

Bağlı liste kullanımı ile kuyruk uygulamasında tamsayı tipinde değerlerin saklanacağıını varsayırsak, ihtiyaç duyduğumuz değişkenleri aşağıdaki gibi listeleyebiliriz:

- *struct Node:* Bağlı listeyi oluşturacak elemanlar için veri yapısı
- *struct Node* front:* Kuyruğun başını ifade eden Node gösterici
- *struct Node* rear:* Kuyruğun sonunu ifade eden Node gösterici

Kuyruk veri yapısının bağlı liste ile uygulanması.

ÖRNEK 2.12

```
#include<stdio.h>
#include<stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* front = NULL;
struct Node* rear = NULL;

void enqueue(int a) {
    struct Node* t = (struct Node*) malloc(sizeof(struct Node));

    t->data = a;
    t->next = NULL;

    if(front == NULL && rear == NULL) {
        front = rear = t;
    }
    else {
        rear->next = t;
        rear = t;
    }

    printf("%d kuyruga eklendi\n", a);
}

void dequeue() {
    if(front == NULL) {
        printf("Kuyrukta eleman yoktur\n");
    }
    else {
        struct Node* t = front;

        if(front == rear) {
            front = rear = NULL;
        }
        else {
            front = front->next;
        }
        printf("%d kuyruktan cikarildi\n", t->data);
        free(t);
    }
}
```

Örnek 2.12'de bağlı liste ile kuyruk uygulamasının nasıl yapılabileceği gösterilmiştir. Kuyrukta tamsayı değerler saklayacak bağlı liste için *Node* adında bir yapı oluşturulmuştur. Bu yapı, *data* adında bir tamsayı değişkenden ve listenin bir sonraki elemanını işaret eden, *next* olarak adlandırılmış *Node* göstéricisinden meydana gelmektedir.

Program içerisinde *front* ve *rear* adlarıyla tanımlanmış *Node* yapıları, bağlı listenin başını ve sonunu işaret etmektedir. Bu değerler tanımlanırken NULL olarak atanmışlardır. Kuyruğa ekleme yapmak için *enqueue()* fonksiyonu, kuyruktan çıkışma yapmak için *dequeue()* fonksiyonu tanımlanmıştır.

Ekleme işleminde *malloc* fonksiyonu ile yeni eleman (*t* adı verilen *Node* gösterici) için hafızada yer açılır. Eklenecek a tamsayı, *t*'nin *data* değişkenine atanır ve *t*'nin *next* göstericisi NULL olarak belirlenir. Bağlı liste boş ise (*front* ve *rear* göstericilerinin her ikisinin de NULL olması durumu), *front* ve *rear* göstericileri *t*'ye eşitlenir. Bağlı liste boş değilse, kuyruğun sonu *t* olacak şekilde tanımlama yapılır.

Çıkarma işleminde öncelikle kuyruğun başı (*front* göstericisinin NULL olması) kontrol edilir ve kuyruğun başında bir eleman varsa işlemeye devam edilir. Kuyruğun başını işaret eden bir gösterici tanımlanır. Bağlı listede tek bir eleman varsa (*front* ve *rear* göstericilerinin eşit olması), *front* ve *rear* göstericileri NULL değerine atanır. Bağlı listede bir den fazla eleman varsa, kuyruğun başı, kuyruğun en baştan ikinci elemanı olacak şekilde tanımlama yapılır. *free* fonksiyonu ile kuyruğunındaki eleman, hafızadan kaldırılır.

SIRA SİZDE



3

Örnek 2.12'de verilen kodu kullanarak, kuyruğa ekleme ve çıkışma işlemleri yapan, kuyruktaki elemanları sırayla ekrana yazan bir program geliştirin. Geliştirdiğiniz programda kuyruğa sırasıyla 5, 12, 9, 1, 7 sayılarını ekleyin, kuyruktan çıkışma işlemini 2 defa uygulayın, kuyruğa tekrar 5 sayısını ekleyin. Kuyruğun son halini ekrana yazın.

DİKKAT



Verilen örnekleri bilgisayar ortamında denemeniz, kuyruklar hakkında deneyiminizin artmasına katkıda bulunacaktır.

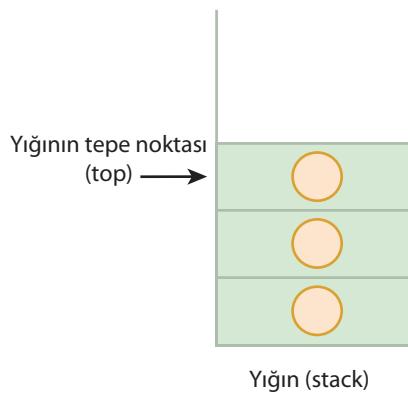
YIĞINLAR

Yaşantımızdaki çeşitli aktivitelerde nesnelerin üst üste dizilmesi gereklidir. Üniversite yemekhanesindeki tepsiler, restoran mutfağındaki tabaklar, elbise dolabı rafındaki kıyafetler, nesnelerin günlük yaşamda üst üste dizilmesi için gösterebilecek basit örneklerdir. Bir üniversite yemekhanesindeki tepsilerden almak istediğimizde, temiz tepsilerin içerişinden en üstte olanı alırız. Temiz tepsiler biriktirilirken, yeni gelen tepsiler var olanların üstüne eklenir. Nesnelerin üst üste dizilişi, günlük hayatı olduğu gibi programlamada da var olan bir gereksinimdir. Bu ihtiyaç, yiğin (stack) adı verilen veri yapıları ile karşılanır.

Yiğin, verilerin doğrusal bir şekilde tutulduğu, ekleme ve çıkışma işlemlerinin en üst noktadan yapıldığı bir veri yapısıdır. Eklenen veri, yiğinin en üst noktasında saklanırken; çıkarılan veri de yiğinin en üst noktasından alınır. Yiğinin en üst noktasının takibi, yiğinin tepe noktası (top) aracılığıyla sağlanır (Şekil 2.9).

Şekil 2.9

Yiğin Veri Yapısının
Mantıksal Gösterimi.



Programlamada, yiğinlar üzerinde yapılan temel işlemler eleman ekleme (push), eleman çıkışma (pop) ve en üstteki elemanı elde etmedir (peek). Pop işleminde en üstte-

ki eleman yiğinden çıkarılırken, peek işleminde yalnızca bu elemanın değeri elde edilir, eleman yiğinden çıkarılmaz. Bu işlemlerin yanı sıra yiğinin doluluk kontrolü (isFull) ve yiğinin boşluk kontrolü (isEmpty) gibi yardımcı fonksiyonlar da kullanılabilir.

Yiğin mantığının bir veri yapısı olarak programlanmasında iki temel yöntem vardır:

- Dizilerin kullanımı ile yiğin programlama
- Bağlı listelerin kullanımı ile yiğin programlama

Dizi ile Yiğin Uygulaması

Yiğin veri yapısını programlamada bir diziden faydalanylabilir. Bu yöntemde verileri tutacak bir diziye ve yiğinin tepe noktasını takip edecek bir tamsayıya ihtiyaç vardır. Yiğin veri yapısının tamsayı tipinde değerleri saklayacağını varsayırsak, programda gerekli olan değişkenleri aşağıdaki gibi listeleyebiliriz:

- int stack[N]*: Yiğindaki elemanları tutacak N uzunluğunda tamsayı dizisi
- int top*: Yiğinin tepe noktasını gösteren indis

Yiğin veri yapısının dizi ile uygulanması.

ÖRNEK 2.13

```
#include<stdio.h>
#define N 10

int stack[N];
int top = -1;

void push(int a) {
    if(top == N-1) {
        printf("Yiginda yer yoktur.\n");
    }
    else {
        stack[++top] = a;
        printf("%d yigina eklendi.\n", a);
    }
}

int pop() {
    if(top < 0) {
        printf("Yiginda eleman yoktur.\n");
        return -1;
    }
    else {
        int a = stack[top --];
        printf("%d yigindan cikarildi.\n", a);
        return a;
    }
}

int peek() {
    if(top < 0) {
        printf("Yiginda eleman yoktur.\n");
        return -1;
    }
    else {
        printf("%d yiginin tepe noktasindadir.\n", stack[top]);
        return stack[top];
    }
}
```

Örnek 2.13'te dizi ile yiğin uygulamasının nasıl yapılabileceği gösterilmiştir. Yiğinda tamsayı verileri saklayacak diziye *stack* adı verilmiş, dizinin boyutu 10 olarak tanımlan-

miştir. Yiğin yapısı için gerekli diğer bir değişken olan *top*, program başında -1 değeri almıştır. Yiğine ekleme yapmak için *push()* fonksiyonu, yiğinden çıkışma yapmak için *pop()* fonksiyonu, yiğinin en tepesindeki elemanı elde etmek için ise *peek()* fonksiyonu tanımlanmıştır.

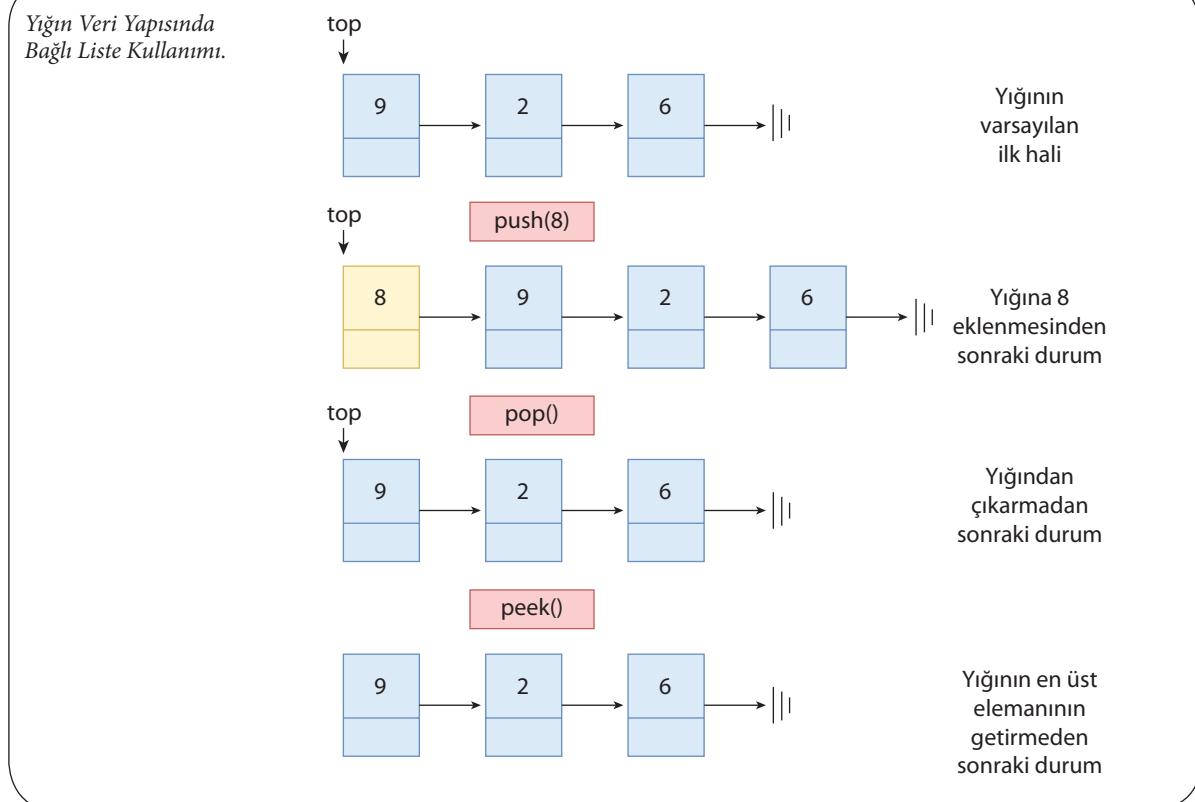
Ekleme işleminde öncelikle yiğinin doluluğu (dizinin son elemanın indisini olan $N-1$ ile *top* değişkenini karşılaştırarak) kontrol edilir ve yiğinda yer varsa işleme devam edilir. Yiğine ekleme işlemi yapılabilecek ise *top* değişkeninin değeri 1 arttırılır, a tamsayısı dizinin *top* değerindeki indisine kaydedilir.

Çıkarma işleminde ve en üstteki elemanı elde ederken benzer işlemler yapılır. Öncelikle yiğinda eleman olup olmadığı kontrol edilir (*top* değişkeni ile 0 değerini karşılaştırarak). Yiğinda eleman varsa, a değişkenine dizinin *top* indisindeki değer atanır. Çıkarmada, *top* değişkeninin değeri 1 azaltılırken, en üstteki elemanı elde edilirken *top* değişkeninin değeri değiştirilmez. Fonksiyon, a değişkenini döndürür.

Bağılı Liste ile Yiğin Uygulaması

Yiğin yapısını programlamak için bir bağlı listeden de faydalabilir. Bu yöntemde bağlı listenin elemanlarını oluşturacak bir veri yapısına ve yiğinin tepe noktasını takip edecek bir göstericiye ihtiyaç duyulur.

Şekil 2.10



Şekil 2.10'da yiğin veri yapısının bağlı liste ile gösterimine yer verilmiştir. Yiğinin ilk halinde 9, 2 ve 6 sayılarını içeren elemanlar bulunmaktadır. Yiğinin tepe noktası, 9 sayısını içeren elemanı göstermektedir. Tanımlanan bu yiğine, öncelikle *push* işlemi ile 8 sayısını içeren bir eleman eklenmiştir ve yiğinin tepe noktası, bu yeni eklenen 8 sayısını içeren elemanı gösterecek şekilde güncellenmiştir. Daha sonraki aşamada, *pop* işlemi ile

yığının en üstündeki eleman yığından çıkarılmıştır ve yığının başı, 9 sayısını içeren elemanın gösterecek şekilde güncellenmiştir. Son olarak, peek işlemi ile yığının en üstündeki elemanın değeri, yani 9 sayısı elde edilmiştir. Bu işlem sonucunda yığının tepe noktasında bir değişiklik olmamıştır.

Yığın veri yapısının bağlı liste ile uygulanması.

ÖRNEK 2.14

```
#include<stdio.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL;

void push(int a) {
    struct Node* t = (struct Node*) malloc(sizeof(struct Node));
    t->data = a;

    if(top == NULL) {
        top = t;
        top->next = NULL;
    }
    else {
        t->next = top;
        top = t;
    }
    printf("%d yigina eklendi\n", a);
}

int pop() {
    if(top == NULL) {
        printf("Yiginda eleman yoktur\n");
        return -1;
    }
    else {
        struct Node* t = top;
        int a = t->data;

        top = top->next;

        printf("%d yigindan cikarildi\n", a);
        free(t);

        return a;
    }
}

int peek() {
    if(top == NULL) {
        printf("Yiginda eleman yoktur\n");
        return -1;
    }
    else {
        printf("%d yiginin tepe noktasindadir\n", top->data);
        return top->data;
    }
}
```

Örnek 2.14'te bağlı liste ile yiğin uygulamasının nasıl yapılabileceği gösterilmiştir. Yiğında tamsayı değerler saklayacak bağlı liste için *Node* adında bir yapı oluşturulmuştur. Bu yapı, *data* adında bir tamsayı değişkenden ve listenin bir sonraki elemanını işaret eden, *next* olarak adlandırılmış *Node* göstericisinden meydana gelmektedir.

Program içerisinde *top* adıyla tanımlanmış *Node* yapısı, bağlı listenin tepe noktasını işaret etmektedir. Bu değerler tanımlanırken NULL olarak atanmışlardır. Yiğine ekleme yapmak için *push()* fonksiyonu, yiğinden çıkarma yapmak için *pop()* fonksiyonu, yiğinin en üstündeki elemanın değerini elde etmek için *peek()* fonksiyonu tanımlanmıştır.

Ekleme işleminde *malloc* fonksiyonu ile yeni eleman (*t* adı verilen *Node* gösterici) için hafızada yer açılır ve eklenecek bir tamsayı, *t*'nin *data* değişkenine atanır. Bağlı liste boş ise (*top* göstericisinin NULL olması), *top* göstericisi *t*'ye eşitlenir. Bağlı liste boş değilse, yiğinin en üzerinde *t* olacak şekilde tanımlama yapılır.

Cıkarma işleminde öncelikle yiğinin tepe noktası kontrol edilir (*top* göstericisinin NULL olması) ve yiğinda herhangi bir eleman varsa işlemeye devam edilir. Yiğinin tepe noktasını işaret eden, *t* adında bir gösterici tanımlanır ve buradaki elemanın değeri a değişkenine atanır. Yiğinin tepe noktası, *t* göstericisinin *next* ile tanımlanmış elemanına kaydırılır. *t* adı ile tanımlanmış eleman *free* fonksiyonu ile hafızadan silinir. Çıkarma fonksiyonu sonuç olarak yiğinden çıkarılan elemanın değeri olan a sayısını döndürür.

En üstteki elemani elde ederken, yiğinin tepe noktası kontrol edilir. Yiğinda herhangi bir eleman varsa, yiğinin en üstünü işaret eden *top* göstericisinin *data* değeri sonuç olarak döndürülür.

DİKKAT



Verilen örnekleri bilgisayar ortamında bizzat denemeniz, yiğinlar hakkındaki deneyiminizi artırmasına katkıda bulunacaktır.

Özet



Tek boyutlu dizi tanımlamayı ve dizinin elemanlarına değer atamayı ifade edebilecek

Dizi, aynı türden verilerin tek bir isim altında tutulmasını sağlayan bir veri yapısıdır. Bir dizinin uzunluğu, dizinin tanımlanması esnasında belirlenir ve bu uzunluk sabit kalır. Dizi tanımlamadaki genel ifade, dizide saklanacak veri tipinden, dizinin adından ve dizinin uzunluğundan meydana gelir.

Dizinin elemanlarına erişim, indis adı verilen sayılar aracılığıyla gerçekleşir. Dizilerin indis değeri 0'dan başlar, dizinin uzunluğunun 1 eksigine kadar artarak devam eder. Dizinin elemanlarına değer atama işlemi, dizi tanımlaması ile birlikte veya dizi elemanlarını indisler aracılığıyla gezinerek yapılabilir.



Çok boyutlu dizi yapılarını açıklamak

Diziler tek boyutlu olabildikleri gibi, birden fazla boyuta da sahip olabilirler. Programlamada en sık kullanılan çok boyutlu diziler, iki ve üç boyutludur. İki boyutlu bir dizi, satır ve sütunlardan oluşan bir tablo şeklinde düşünülebilir. Üç boyutlu bir dizi ise iki boyutlu dizilerin dizisi olarak tanımlanabilir.

Çok boyutlu dizilerde, dizi elemanlarını gezinmek için dizinin tüm boyutlarını dolaşmak gereklidir. Bu gereksinim için iç içe geçmiş for döngüleri kullanılabilir.



Bağlı liste yapısını tanımlamak ve bağlı liste çeşitlerini sıralamak

Bağlı liste (linked list), aynı türden nesnelerin doğrusal bir sırada ve birbirlerine bağlı şekilde saklandığı veri yapısıdır. Bağlı listedeki nesnelere düğüm (node) denir. Genellikle iki kısımdan oluşan düğümler, birbirlerine göstericiler aracılığıyla bağlanır. Ayrıca, bağlı listelerde listenin başlangıcı işaret eden bir gösterici (head pointer) de bulunur.

Bağlı listeler, liste içerisindeki hareket kabiliyetine göre 3 temel kategoriye ayrılır. Sadece ileri yönde hareket edilebilen listeler tek yönlü, hem ileri hem de geri hareket edilebilen listeler çift yönlü, liste sonundan liste başına geçiş yapılabilen listeler ise dairesel olarak tanımlanır.



Kuyruk yapısının mantığını açıklamak ve temel kuyruk işlemlerini sıralamak

Kuyruk (queue), verilerin doğrusal bir sırada saklandığı, kuyruk başının (front) ve sonunun (rear) takip edildiği bir veri yapısıdır.

Kuyruğa eleman eklemeye (enqueue) ve kuyruktan eleman çıkarmaya (dequeue), kuyruk veri yapısındaki temel işlemlerdir. Eklenen elemanlar kuyruk sonuna kaydedilirken, çıkarılan elemanlar ise kuyruk başından alınır. Dolayısıyla kuyruğa giren ilk eleman, kuyruktan da ilk çıkar. Bu kural, programlamada FIFO (First-In First-Out) olarak anılır.



Dizilerle ve bağlı listelerle kuyruk veri yapısını oluşturmak

Kuyruk mantığını programlamak için bir diziden veya bir bağlı listeden faydalanylabilir. Dizi kullanımı ile kuyruk programlarken, kuyruğun mevcut eleman sayısını takip etmek ve diziyi dairesel bir yapıya oturtmak gereklidir. Bağlı liste kullanımı ile kuyruk programlamada ise kuyruğun eleman sayısının takibi gereklidir, ancak yazılan kod daha karmaşıktır.



Yığın yapısının mantığını kavramak ve temel yığın işlemlerini sıralamak

Yığın (stack), eleman eklemeye ve çıkış işlemlerinin en üst noktadan yapıldığı veri yapısıdır. Yığın veri yapısında yığının tepe noktası (top) takip edilir.

Yığın için tanımlı temel işlemler eleman eklemeye (push), eleman çıkışına (pop) ve en üstteki elemanın değerini getirmeye (peek). Yığında eklemeye ve çıkış işlemleri her zaman yığının en üstünden gerçekleşir. Dolayısıyla yığına son giren eleman, yığından da ilk çıkar. Bu kural, programlamada LIFO (Last-In First Out) olarak adlandırılır.



Dizilerle ve bağlı listelerle yığın veri yapısını oluşturmak

Yığınların programlamasında bir dizi veya bir bağlı liste kullanmak mümkündür. Dizi ile yığın programlamada yığının üst noktası bir indis değeri ile takip edilirken, bağlı liste ile yığın programlamada yığının üst noktasını işaret eden bir gösterici kullanılır.

Kendimizi Sınavalım

1. Aşağıdaki seçeneklerden hangisi 10 elemanlı bir dizinin ilk ve son indislerine ait numaraları göstermektedir?
 - a. 0 ve 1
 - b. 0 ve 9
 - c. 0 ve 10
 - d. 1 ve 9
 - e. 1 ve 10

2. Aşağıdaki dizi tanımlamalarından hangisi derleyici tarafından derlendiğinde bir hata meydana gelir?
 - a. double dizi[3];
 - b. double dizi[3] = {};
 - c. int dizi[] = {1, 2, 3};
 - d. int dizi[4] = {1, 2, 3};
 - e. int dizi[4] = {0, 1, 2, 3, 4};

3. Aşağıdakilerden hangisi diziler ve bağlı listelerin farklılığı gösterdiği durumlardan **değildir**?
 - a. Veri ekleme maliyeti
 - b. Veri çıkışma maliyeti
 - c. Hafıza kullanımı
 - d. Değişken isimlendirme
 - e. Verilere doğrudan erişim

4. Bir bağlı listede, listenin en sonundan en başına geçibilme imkanı varsa, o bağlı listenin hangi alt kategoriye ait olduğu söylenir?
 - a. Tek yönlü bağlı liste
 - b. Çift yönlü bağlı liste
 - c. Dairesel bağlı liste
 - d. Serbest bağlı liste
 - e. Dizi yapısında bağlı liste

5.

```
#include<stdio.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* head = NULL;

void traverse() {
    struct Node *t = head;

    while(t != NULL) {
        printf("%d ", t->data);
        t = t->next;
    }
}
```

Tek yönlü bir bağlı liste için hazırlanmış yukarıdaki koda göre, traverse fonksiyonun görevi nedir?

- a. Listenin elemanlarını dolaşıp, ekrana yazmak
- b. Listeye yeni eleman eklemek
- c. Listenin ilk elemanını silmek
- d. Listenin son elemanını silmek
- e. head göstericisini sıfırlamak

6. Aşağıdaki veri yapılarından hangisi, "İlk giren, ilk çıkar." şeklinde tercüme edilen FIFO (First-In First-Out) kuralı ile birlikte anılmaktadır?

- a. Dizi
- b. Tek yönlü bağlı liste
- c. Çift yönlü bağlı liste
- d. Yığın
- e. Kuyruk

7. Dizgi tipinde verileri saklayacak şekilde tanımlanmış bir kuyruğa "Java", "Ruby", "Perl", "Python", "JavaScript", ve "C++" dizgileri yazılış sırasıyla ekleniyor. Sonrasında dequeue işlemi ile kuyruktan 2 eleman çıkarılıyor. Oluşan son durumda, kuyruğun başında hangi eleman yer alır?

- a. C++
- b. Perl
- c. Ruby
- d. JavaScript
- e. Python

8. Aşağıdaki veri yapılarından hangisi "Son giren, ilk çıkar." şeklinde tercüme edilen LIFO (Last-In First-Out) kuralı ile birlikte anılır?

- a. Yığın
- b. Kuyruk
- c. Dizi
- d. Çift yönlü bağlı liste
- e. Dairesel bağlı liste

9.

```
struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL;

int peek() {
    if(top == NULL) {
        return -1;
    } else {
        return top->data;
    }
}
```

Yığın veri yapısı için hazırlanmış yukarıdaki koda göre, peek fonksiyonun işlevi nedir?

- a. Yığına yeni bir eleman eklemek
- b. Yığının en üzerindeki elemanı silmek
- c. Yığında bulunan elemanları ekrana yazdırma
- d. Yığının tepe noktasını bir aşağı kaydirmak
- e. Yığının tepe noktasındaki elemanın değerini döndürmek

10. Tamsayı tipinde verileri saklayacak şekilde tanımlanmış bir yığına 4, 5, 2, 1, 3 sayıları yazılış sırasıyla eklendiğinde, yığının tepe noktasında hangi eleman yer alır?

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

Kendimizi Sınavalım Yanıt Anahtarı

1. b Yanınız yanlış ise “Diziler” konusunu yeniden gözden geçiriniz.
2. e Yanınız yanlış ise “Diziler” konusunu yeniden gözden geçiriniz.
3. d Yanınız yanlış ise “Bağlı Listeler” konusunu yeniden gözden geçiriniz.
4. c Yanınız yanlış ise “Bağlı Listeler” konusunu yeniden gözden geçiriniz.
5. a Yanınız yanlış ise “Bağlı Listeler” konusunu yeniden gözden geçiriniz.
6. e Yanınız yanlış ise “Kuyruklar” konusunu yeniden gözden geçiriniz.
7. b Yanınız yanlış ise “Kuyruklar” konusunu yeniden gözden geçiriniz.
8. a Yanınız yanlış ise “Yığınlar” konusunu yeniden gözden geçiriniz.
9. e Yanınız yanlış ise “Yığınlar” konusunu yeniden gözden geçiriniz.
10. c Yanınız yanlış ise “Yığınlar” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

Aşağıda gösterilen program, 5 elemanlı bir tamsayı dizisine kullanıcının veri girişi yapmasını sağlar. Kullanıcı tarafından girilen değerler, program çıktısı olarak ekrana yazdırılır.

```
#include <stdio.h>
#define N 5

void yazdir(int a[]) {
    int i;

    for(i=0; i<N; i++) {
        printf("%d. sayı = %d\n", i+1, a[i]);
    }
}

int main(void) {
    printf("Lutfen %d adet tamsayı giriniz:\n", N);

    int dizi[N];
    int i;

    for(i=0; i<N; i++) {
        scanf("%d", &dizi[i]);
    }

    yazdir(dizi);
    getch();

    return 0;
}
```

Sıra Sizde 2

Bu alıştırmada sizden bekleneni karşılamak için, aşağıdaki ana fonksiyonu yazmanız ve örnekte belirtilen kod ile birlikte çalıştırmanız gereklidir.

```
int main(void) {

    enqueue(5);
    enqueue(12);
    enqueue(9);
    enqueue(8);
    enqueue(0);
    enqueue(1);
    enqueue(7);

    dequeue();
    dequeue();

    enqueue(5);

    getchar();
    return 0;
}
```

Sıra Sizde 3

Bu alıştırmada sizden beklenen, Örnek 2.12'de verilen programa uygun bir ana fonksiyon ve kuyruk içeriğini ekrana yazdırın bir fonksiyon yazmakta. Aşağıda verilen program, sizden istenilenleri karşılamaktadır.

```
#include<stdio.h>
#include<stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

struct Node* front = NULL;
struct Node* rear = NULL;

void enqueue(int a) {
    struct Node* t = (struct Node*) malloc(sizeof(struct Node));
    t->data = a;
    t->next = NULL;
    if(front == NULL && rear == NULL) {
        front = rear = t;
    } else {
        rear->next = t;
        rear = t;
    }
    printf("%d kuyruga eklendi.\n", a);
}

void dequeue() {
    if(front == NULL) {
        printf("Kuyrukta eleman yoktur.\n");
    } else {
        struct Node* t = front;
        if(front == rear) {
            front = rear = NULL;
        } else {
            front = front->next;
        }
        printf("%d kuyruktan cikarildi.\n", t->data);
        free(t);
    }
}

void printQueue() {
    printf("%Kuyruktaki elemanlar: ");
    struct Node* temp = front;
    while(temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main(void) {
    enqueue(5);
    enqueue(12);
    enqueue(9);
    enqueue(1);
    enqueue(7);

    dequeue();
    dequeue();

    enqueue(5);
    printQueue();
    getch();
    return 0;
}
```

Yararlanılan ve Başvurulabilecek Kaynaklar

- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) **Introduction to Algorithms**, 3rd Edition, MIT Press.
- Levitin A. (2012) **Introduction to The Design & Analysis of Algorithms**, 3rd Edition, Pearson.
- Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

3

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Ağaç veri yapısında yer alan temel kavramları, ikili ağaçları ve ikili arama ağaçlarını tanımlayabilecek,
 - 🕒 Ağaçlar üzerinde gezinme yöntemlerini sıralayabilecek,
 - 🕒 AVL ağacı veri yapısını tanımlayabilecek ve ağaç dengesinin korunması için gerekli döndürme işlemlerini açıklayabilecek,
 - 🕒 Yiğin ağaçlarının temel özelliklerini ve bu veri yapısında ekleme çıkarma işlemleriini özetleyebilecek,
 - 🕒 Özetleme tablosu veri yapısını ve özetleme fonksiyonlarını tanımlayabilecek,
 - 🕒 Özetleme fonksiyonlarında meydana gelen çalışmaları çözümleme yöntemleri ni listeleyebilecek
- bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- İkili Ağaç
- İkili Arama AĞacı
- AVL Ağacı
- Yiğin Ağacı
- Özetleme Tablosu
- Özetleme Fonksiyonu
- Çatışma
- Ayrik Zincirleme
- Açık Adresleme

İçindekiler

Algoritmalar ve Programlama

Ağaçlar, Yiğin Ağaçları
ve Özetleme Tabloları

- GİRİŞ
- AĞAÇLAR
- YIĞIN AĞACLARI
- ÖZETLEME TABLOLARI

Ağaçlar, Yığın Ağaçları ve Özetteleme Tabloları

GİRİŞ

Kitabımızın bu ünitesinde ağaçlar, yığın ağaçları ve özetteleme tabloları anlatılmaktadır.

“Ağaçlar” başlığında ağaç veri yapısının tanımı yapılmış ve başlıca ağaç türleri olan ikili ağaçlar, ikili arama ağaçları ve AVL ağaçları anlatılmıştır. “Yığın Ağaçları (Heap)” başlığında özel bir ağaç yapısı olan yığın ağaçları gösterilmiş ve bu veri yapısının temel özelliklerinden bahsedilmiştir. Ünitenin son başlığı olan “Özetteleme (Hash) Tabloları” bölümünde ise, verilerin anahtar ve veri çifti şeklinde saklanmasına imkan veren özetteleme tabloları gösterilmiştir.

Ünite genelinde verilen birçok örnek ve alıştırma, konuya ilginizi ve konu genelindeki hakimiyetinizi artırmayı amaçlamaktadır. Örnek programları bilgisayar ortamında denemeniz ve geliştirmeniz, veri yapıları konusunda pratiğinizi ve bilginizi artıracaktır.

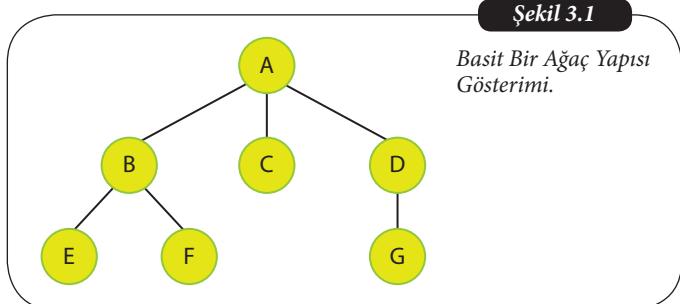
AĞAÇLAR

Ağaç veri yapısı, verilerin birbirlerine temsili bir ağaç oluşturacak şekilde bağlandığı hiyerarşik bir veri modelidir. Bir ağaç düğümlerden ve düğümleri birbirine bağlayan dallardan meydana gelir.

Ağaç veri yapısı, çizge veri yapısının bir alt kümesidir. Bir çizgenin ağaç olabilmesi için, her iki düğüm arasında sadece bir yol olmalı, düğümler arasındaki yolda döngü (cycle) olmamalıdır.

Ağaç veri yapısında bilinmesi gereken başlıca kavramlar aşağıda listelenmiştir:

- Kök (Root): Bir ağacın en üst noktasında bulunan düğümdür.
- Dal (Edge): Düğümleri birbirine bağlayan kenara verilen isimdir.
- Yol (Path): Birbirleri ile bağlantılı dal dizisine yol adı verilir.
- Yol Uzunluğu (Length of a Path): Bir yolu oluşturan dal dizisindeki dal sayısıdır.
- Ebeveyn (Parent): Bir düğümden önce yer alan ve o düğüme bir dal ile bağlı olan düğüme ebeveyn denir. Kök hariç her düğümün bir ebeveyni bulunmaktadır.
- Çocuk (Child): Bir düğümden sonra yer alan ve o düğüme bir dal ile bağlı olan düğüm/düğümlere çocuk denir.
- Ağaç Yüksekliği (Height of a Tree): Bir ağacın kökünden ağaçtaki en alt çocuğa kadar olan yoluun uzunluğudur.
- Düğüm Yüksekliği (Height of a Node): Bir düğümden ağaçtaki en alt çocuğa kadar olan yoluun uzunluğudur.



- Düyüküm Derinliği (Depth of a Node): Bir düğümden ağaç köküne kadar olan yolun uzunluğuudur.

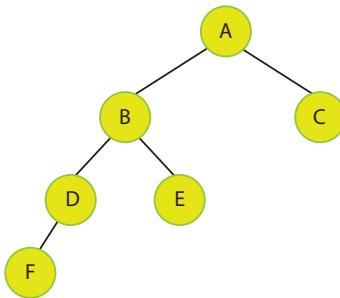
Şekil 3.1'de basit bir ağaç yapısı gösterilmiştir. Bu ağacın kökü A düşümüdür. B, C ve D düşümleri ise A'nın çocuklarıdır. E ve F düşümleri B'nin çocukları, G düşümü ise D'nin çocuğuudur. C, E, F ve G düşümlerinin herhangi bir çocuğu bulunmamaktadır. Kök düşüm olan A'dan en alt çocuklara uzanan yolun uzunluğu 2 olduğu için, bu ağacın yüksekliği de 2'dir. F düşümünün yüksekliği 0, derinliği ise 2'dir.

Bir ağaç veri yapısı, sahip olduğu özellikler ile farklı kategorilere ayrılabilir. Kitabımızda ağaç veri yapısı başlığı altında ikili ağaçlar, ikili arama ağaçları ve AVL ağaçları işlenecektir.

İkili Ağaçlar (Binary Trees)

Şekil 3.2

Örnek Bir İkili Ağaç Gösterimi.



İkili ağaçlar, her bir düşümün en fazla 2 çocuğu sahip olabileceği ağaç türüdür. Bu veri yapısında ekleme, silme ve arama işlemleri çok hızlı bir şekilde yapılmaktadır.

Şekil 3.2'de gösterilen ağaçta A ve B düşümleri 2'ser çocuğu, D düşümünün 1 çocuğu vardır. C, E ve F düşümlerinin ise herhangi bir çocuğu yoktur. Bu ağaçtaki her bir düşümün en fazla 2 çocuğu bulunduğu için, gösterilen ağaç ikili ağaç sınıfına girer. Şekil 3.1'de gösterilen ağaçta ise, kök olan A düşümünün 3 çocuğu bulunmaktadır. Dolayısıyla, bu ağacın ikili ağaç olduğu söylenemez.

ÖRNEK 3.1

İkili ağaç'a düğüm ekleyen ve düğümleri ekrana yazan program.

```

#include<stdio.h>

typedef struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;

TreeNode* createNode(int x) {
    TreeNode* t = (TreeNode*) malloc(sizeof(TreeNode));
    t->data = x;
    t->left = NULL;
    t->right = NULL;
    return t;
}

void printTree(TreeNode *root) {
    if(root != NULL) {
        printf("%d ", root->data);
        printTree(root->left);
        printTree(root->right);
    }
}

int main(void) {
    TreeNode *root = createNode(5);
    root->left = createNode(6);
    root->right = createNode(8);

    printTree(root);
    getch();
    return 0;
}
  
```

Örnek 3.1'de *struct* kullanımı ile ikili ağaç veri yapısının nasıl tanımlanabileceği gösterilmiştir. Tanımlanan ağaç yapısındaki bir düğüm *int* veri tipinde bir değere, *left* olarak adlandırılmış *TreeNode* göstericisine ve *right* olarak adlandırılmış *TreeNode* göstericisine sahiptir. *left* ve *right* olarak isimlendirilen göstericiler, düğümün sol ve sağ çocuklarını ifade etmektedir. Örnekteki *createNode()* fonksiyonunun görevi *TreeNode* tipinde bir düğüm oluşturmak, *printTree()* fonksiyonunun görevi ise ağaçta tanımlı düğümleri gezerek düğüm değerlerini ekrana yazdırmaktır. Ana fonksiyon içerisinde 5’i içeren bir ağaç düğümü oluşturulmuş ve bu düğüm kök olarak belirlenmiştir. Kökün sol çocuğuna 6’ı içeren bir düğüm, sağ çocuğuna da 8’i içeren bir düğüm yerleştirilmiştir. Bu örnekteki program çalıştırıldığında, ağaçta yer alan düğümler gezilerek ekrana yazdırılacak, sonuç olarak ekranda “5 6 8” değerleri gözükecektir.

İkili Ağaçlarda Gezinme Yöntemleri

Bir ağacın düğümlerini belirli bir algoritma ve sira çerçevesinde dolaşma eylemine ikili ağaçta gezinme adı verilir. Bir bilgisayar programındaki ağaç veri yapısında gezinmenin düğümlerde arama yapma, düğümleri kullanıcıya gösterme, düğüm değerlerini ekrana yazdırma gibi çeşitli sebepleri olabilir.

İkili ağaç veri yapısı kendi içerisinde alt ağaçlardan meydana geldiği için, ikili ağaçları gezinmede özyinelemeli fonksiyonlar kullanılır. İkili ağaçlardaki düğümler dolaşılırken farklı yöntemler uygulanabilirken, bilgisayar programında bu işi yapabilmek için kabul görmüş üç gezinme yöntemi bulunmaktadır:

- i. Preorder Gezinme (Kök başta)
- ii. Inorder Gezinme (Kök ortada)
- iii. Postorder Gezinme (Kök sonda)

Preorder Gezinme

Bu yöntemde öncelikle kök, daha sonrasında sol alt ağaç, en son olarak da sağ alt ağaç üzerinde gezinme yapılır. Bu yöntemi akılda tutmak için “Root – Left – Right” terimini kullanabiliriz.

Preorder gezinme yöntemi uygulayan özyinelemeli fonksiyon.

ÖRNEK 3.2

```
void preorder(TreeNode *root) {
    if(root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
```

Örnek 3.2'de preorder gezinme yöntemini uygulayan özyinelemeli fonksiyon gösterilmiştir. Şekil 3.2'deki ikili ağaç bu yöntemle gezildiğinde ekrana yazdırılan düğüm değerleri sırasıyla A, B, D, F, E, C olur.

Inorder Gezinme

Bu yöntemde öncelikle sol alt ağaç, daha sonrasında kök, en son olarak da sağ alt ağaç üzerinde gezinme yapılır. Bu yöntemi akılda tutmak için “Left – Root – Right” terimini kullanabiliriz.

ÖRNEK 3.3

Inorder gezinme yöntemi uygulayan özyinelemeli fonksiyon.

```
void inorder(TreeNode *root) {
    if(root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
```

Örnek 3.3'te inorder gezinme yöntemini uygulayan özyinelemeli fonksiyon gösterilmiştir. Şekil 3.2'deki ikili ağaç bu yöntemle gezildiğinde ekrana yazdırılan düğüm değerleri sırasıyla F, D, B, E, A, C olur.

Postorder Gezinme

Bu yöntemde öncelikle sol alt ağaç, daha sonrasında sağ alt ağaç, en son olarak da kök üzerinde gezinme yapılır. Bu yöntemi akılda tutmak için “Left – Right – Root” terimini kullanabiliriz.

ÖRNEK 3.4

Postorder gezinme yöntemi uygulayan özyinelemeli fonksiyon.

```
void postorder(TreeNode *root) {
    if(root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
```

Örnek 3.4'te postorder gezinme yöntemini uygulayan özyinelemeli fonksiyon gösterilmiştir. Şekil 3.2'deki ikili ağaç bu yöntemle gezildiğinde ekrana yazdırılan düğüm değerleri sırasıyla F, D, E, B, C, A olur.

SIRA SİZDE



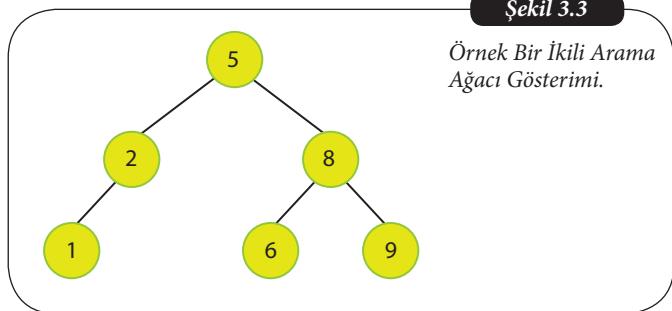
İkili ağaçlar üzerinde gezinme yöntemlerini pekiştirmek için Örnek 3.1'deki programa üc temel gezinme yöntemini ekleyiniz. Ana fonksiyon içerisinde 5 elemanlı bir ağaç oluşturup, oluşan ağaçtaki değerleri tüm gezinme yöntemlerini kullanarak ekrana yazdırınız.

İkili Arama Ağaçları (Binary Search Trees)

İkili arama ağaçları, ikili ağaçların özel bir türüdür. Bu veri yapısında, ikili ağaç özelliklerine ek olarak düğümlerde yer alan veriler arasında büyüklük-küçüklik ilişkisi bulunmaktadır.

İkili ağaç özellikleri taşıyan bir ağacın ikili arama ağaç olabilmesi için ağaçtaki her düğümün, sol alt ağacındaki tüm değerlerden büyük olması, sağ alt ağacındaki tüm değerlerden küçük veya eşit olması gerekmektedir.

Şekil 3.3'te gösterilen ağaçtaki her bir düğümün en fazla 2 çocuğu bulunmaktadır. Dolayısıyla bu ağaç, ikili ağaç sınıfına girmektedir. Düğümlerde yer alan veriler incelediğinde, her bir düğümdeki değerin düğüme ait sol alt ağaçtaki değerlerden büyük, sağ alt ağaçtaki değerlerden küçük olduğu görülmektedir. İkili ağaç özelliklerini taşıyan ve içeriği veriler arasında büyülük-küçüklik kuralını sağlayan bu ağaç, ikili arama ağaçıdır.



Şekil 3.3

Örnek Bir İkili Arama Ağaçları Gösterimi.

İkili arama ağaçlarında inorder gezinme yöntemi uygulandığında, veriler küçükten büyüğe doğru sıralanmış şekilde gezilmiş olur.

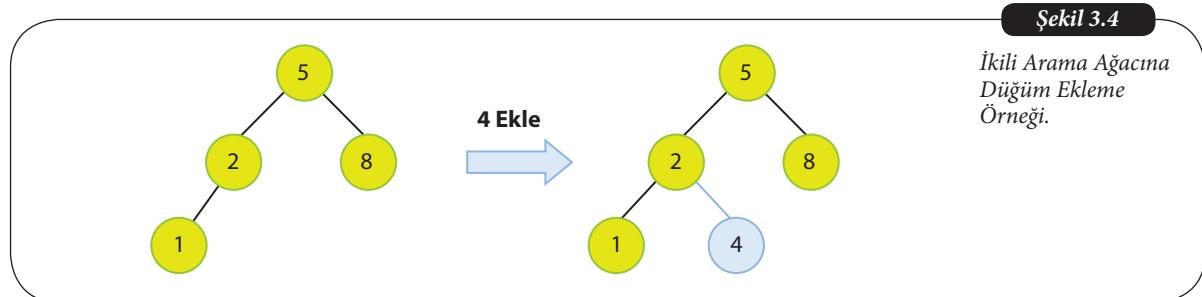


DİKKAT

İkili Arama Ağaçına Düğüm Ekleme

İkili arama ağaçına düğüm ekleme işlemi, ikili ağaçta düğüm eklemekten daha karmaşıktır. İkili arama ağaçına düğüm eklenirken, ikili arama ağaçları veri yapısının özellikleri korunmalıdır.

İkili arama ağaçına düğüm eklemeye, ağaçın düğümlerinin değerleri arasındaki büyülük-küçüklik ilişkisini korumak için eklenecek düğümün yeri tespit edilmelidir. Ağaç boş ise yeni düğüm, ağaçın kökü olarak tayin edilir. Ağaç dolu ise ağaçın kökünden yola çıkılarak, eklenecek düğümün değeri ile kök düğümün değeri karşılaştırılır. Yeni düğümün değeri köktekiden büyükse sağ alt ağaç, küçükse sol alt ağaç doğrulara ilerlenir. Bu işlem, yeni düğüm için uygun bir yer bulunan kadar tekrarlanır.



Şekil 3.4

İkili Arama Ağaçına Düğüm Ekleme Örneği.

Şekil 3.4'te bir ikili arama ağaçına yeni bir düğüm ekleme örneği gösterilmiştir. Bu örnekte var olan ağaçta, 4 değerine sahip yeni bir düğüm eklenmektedir. Ekleme işleminde öncelikle kök düğümün değerine bakılır. Kök düğümün değeri 5 olup bu değer 4'ten büyüktür. Dolayısıyla yeni düğümü eklemek için sol alt ağaç gidilir. Sol alt ağaçın kökünden yer alan düğümün değeri 2 olup bu değer 4'ten küçüktür. Yeni düğümü eklemek için sağ alt ağaç gidilir, sağ alt ağaç boş olduğu için yeni düğüm buraya eklenir. İkili arama ağaçını temsil edecek veri yapısı ve ağaçta düğüm ekleme fonksiyonu Örnek 3.5'te gösterilmiştir.

ÖRNEK 3.5

İkili arama ağacı yapısı ve düğüm ekleme fonksiyonu.

```
typedef struct TreeNode {
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

TreeNode* insertNode(TreeNode *node, int x) {
    if(node == NULL) {
        TreeNode *t = (TreeNode *) malloc(sizeof(TreeNode));
        t -> data = x;
        t -> left = t -> right = NULL;
        return t;
    }
    else if(x > node->data) {
        node->right = insertNode(node->right, x);
    }
    else {
        node->left = insertNode(node->left, x);
    }
}
```

İkili Arama Ağacından Düğüm Çıkarma

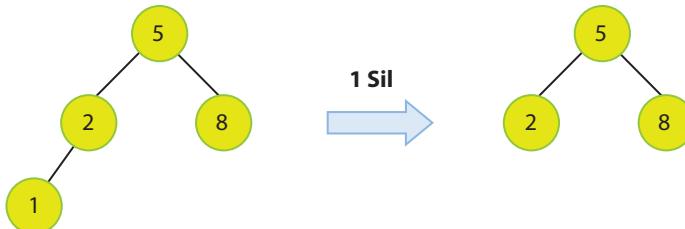
İkili arama ağacından düğüm çıkarma için öncelikle düğümün ağaçta bulunması gereklidir. Düğüm ağaçta yer alıysa, ikili arama ağacı özellikleri korunarak çıkışma işlemi gerçekleştirilebilir.

İkili arama ağacından düğüm çıkarırken incelenmesi gereken üç durum vardır:

- Çıkarılacak düğümün çocuğu yok ise; düğümün ebeveyninin ilgili göstericisi (*left* veya *right*) NULL yapılır, düğüm hafızadan silinir.

Şekil 3.5

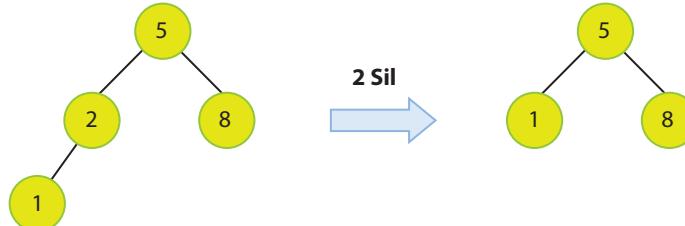
İkili Arama
Ağacından Çocuğu
Olmayan Düğümü
Çıkarma Örneği.



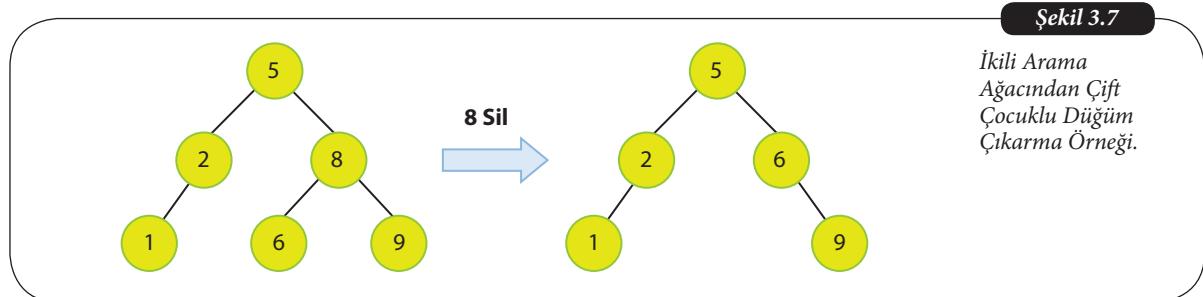
- Çıkarılacak düğümün 1 çocuğu var ise; düğümün çocuğundan itibaren olan alt ağaç düğümün ebeveynine bağlanır, düğüm hafızadan silinir.

Şekil 3.6

İkili Arama
Ağacından Tek
Çocuklu Düğümlü
Çıkarma Örneği.



- iii. Çıkarılacak düğümün 2 çocuğu var ise düğümün sağ alt ağacındaki en küçük değerli düğüm bulunur, bulunan düğüm ile çıkarılacak düğüm yer değiştirilir, düğüm hafızadan silinir.



Şekil 3.5'te gösterilen örnekte ağaçtan 1 değerine sahip düğüm silinmiştir. Bu düğümün herhangi bir çocuğu olmadığı için, düğüm doğrudan ağaçtan çıkarılabilir.

Şekil 3.6'da gösterilen örnekte 2 değerine sahip düğüm silinmiştir. Bu düğümün bir çocuğu olduğu için, düğümün çocuğu olan 1 değerine sahip düğüm, düğümün ebeveyni olan 5 değerine sahip düğüme bağlanmıştır.

Şekil 3.7'de gösterilen örnek, ikili arama ağacından düğüm çıkarma işlemlerinin en karmaşık olanağıdır. Ağaçtan çıkarılacak 8 değerine sahip düğümün iki çocuğu bulunmaktadır. Bu yüzden, bu düğümün sağ alt ağacında yer alan, en küçük değere sahip düğüm bulunmalıdır. Çıkarılacak düğümün sağ alt ağacındaki en küçük değerli düğüm, 6 değerine sahip düğümdür. Bu düğüm çıkarılacak düğüm ile yer değiştirilir. 6 değerine sahip düğüm 8 değerine sahip düğüm yerine geçer. Daha sonrasında ise 8 değerine sahip düğüm ağaçtan çıkarılır.

Bu bölümde anlatılan ikili arama ağaçlarından düğüm çıkarma durumlarını göz önüne alarak, ikili arama ağaçlarından düğüm çıkarma fonksiyonunu geliştiriniz. Yazacağınız program için Örnek 3.5'i temel alıp, bu örnek üzerinden kod geliştirmeye çalışırsınız. Programınızdaki ağaçta düğüm ekleme, ağaçtan düğüm çıkarma ve ağaç ekrana yazdırma fonksiyonlarını deneyip, sonuçlarını gözlemleyiniz.



SIRA SİZDE

AVL Ağaçları

İkili ağaçlarda ve ikili arama ağaçlarında ağaçın yüksekliği için herhangi bir ölçüt bulunmamaktadır. N adet düğüme sahip bir ikili ağaçın yüksekliği en fazla $N-1$ olabilir. Bu ağaçlarda yükseklik için bir kısıtlama olmaması, ağaç içerisindeki düğümlerin dengesiz dağılmasına sebep olabilir. Başka bir ifadeyle, sol alt ağaç ile sağ alt ağaç arasındaki yükseklik farkı 1'den fazla olabilir.

AVL (Adelson – Velsky – Landis) ağaçları, ikili arama ağaçlarının özel bir türüdür. Bu veri yapısında ağaç içerisindeki denge korunmakta, sol alt ağaç ile sağ alt ağaç arasındaki yükseklik farkı en fazla 1 olabilmektedir.

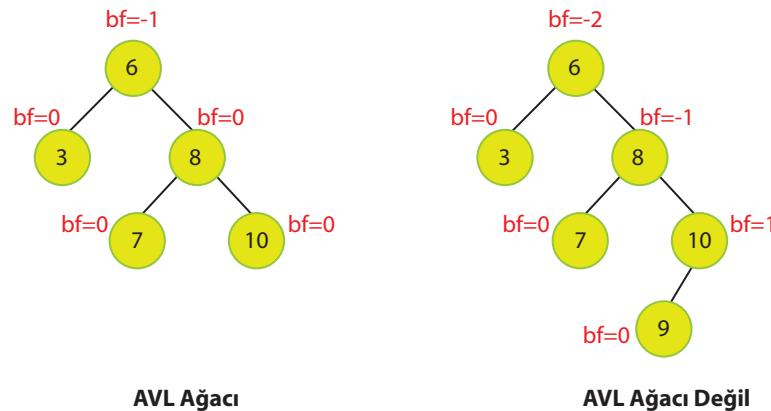
AVL ağaçlarındaki düğümler için **denge faktörü** aşağıdaki formül ile hesaplanır ve dengeli bir ağaç için bu değerler yalnızca -1, 0 ve 1 olabilir:

$$b_f = h_{Left} - h_{Right}$$

Denge Faktörü (Balance Factor): Bir düğümün sol alt ağacının yüksekliği ile sağ alt ağacının yüksekliği arasındaki farka denge faktörü adı verilir.

Sekil 3.8

AVL Ağacı Olan ve
AVL Ağacı Olmayan
Ağaç Örnekleri.



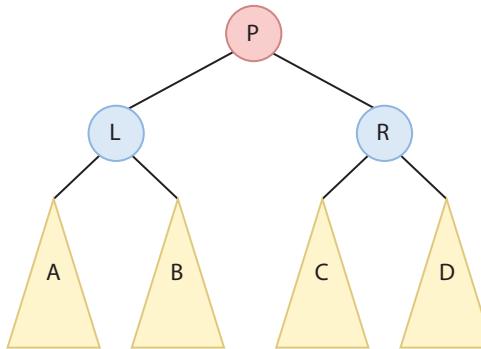
Pivot: Denge faktörü 2 veya -2 olan düğüme pivot adı verilir. AVL ağaçlarında pivot düğüm üzerinde döndürmeler yapılarak denge sağlanır.

Şekil 3.8'de iki adet ikili arama ağacı gösterilmiştir. Her iki ağaç için de denge faktörleri hesaplanmış, ilgili düğümün üzerinde kırmızı ile belirtilmiştir. Şekilde verilen ilk ağaçtaki tüm denge faktörleri 0 ve -1 değerlerinden oluşmaktadır. Dolayısıyla ilk ağaç, bir AVL ağacıdır. Şekildeki ikinci ağaçtaki denge faktörleri ise -1, 0, 1 ve -2 değerlerinden oluşmaktadır. Kökte yer alan, 6 değerini taşıyan düğümün denge faktörü -2 olduğu için bu ağaç bir AVL ağacı değildir.

AVL ağaçları için düğüm ekleme ve düğüm çıkarma işlemleri, ağaçtaki düğümlerin dengesini bozılmamaktadır. Dolayısıyla bu işlemlerde ağacın dengesini korumak için **pivot** düğüm üzerinde çeşitli döndürmeler yapılır.

Sekil 3.9

AVL Ağacında Pivot
ve Sol-Sağ Alt Ağaçlar.

pivot ($bf=2$ veya -2)

Şekil 3.9'da dengesiz durumdaki bir AVL ağacının pivotu (P), sol alt ağaç (L) ve sağ alt ağaç (R) gösterilmiştir. Şeklin alt tarafında yer alan A, B, C ve D üçgenleri ise sol ve sağ alt ağaç için süregelen alt ağaçlardır. AVL ağaçları için ekleme ve çıkarma işlemleri anlatılırken aşağıdaki terminolojiden faydalanailecektir:

- P: Pivot
- L: Pivotun sol alt ağaçları
- R: Pivotun sağ alt ağaçları

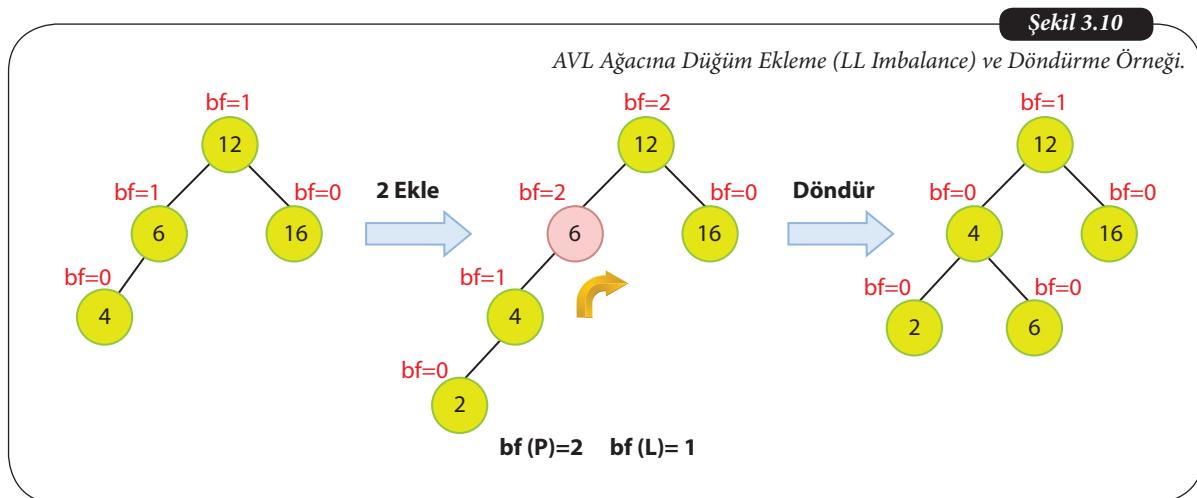
- A bölgesi: Pivotun sol alt ağacının sol çocuğu.
- B bölgesi: Pivotun sol alt ağacının sağ çocuğu.
- C bölgesi: Pivotun sağ alt ağacının sol çocuğu.
- D bölgesi: Pivotun sağ alt ağacının sağ çocuğu.

AVL Ağacına Düğüm Ekleme ve Denge Korunuğu

AVL ağacına düğüm eklerken, öncelikle düğümün ekleneceği yer bulunur. Düğüm eklen dikten sonra oluşan yeni ağaç üzerinde denge faktörleri hesaplanır, ağaçta bir dengesizlik olması durumunda dengesizliğin olduğu düğüm (pivot) tarafında bir veya iki tane döndürme işlemi uygulanır.

Ekleme işleminden sonra oluşan dengesizlik, dört ayrı şekilde görülebilir:

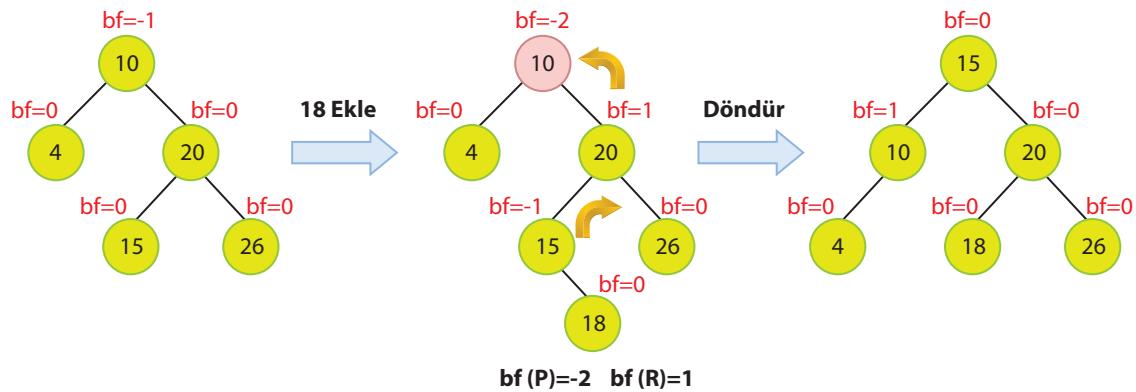
- A bölgesine ekleme (LL Imbalance): P'nin denge faktörü 2, L'nin denge faktörü 0 veya 1 iken karşılaşılır, pivot etrafında sağa doğru tek döndürme ile çözülür.
- D bölgesine ekleme (RR Imbalance): P'nin denge faktörü -2, R'nin denge faktörü değeri 0 veya -1 iken karşılaşılır, pivot etrafında sola doğru tek döndürme ile çözülür.
- C bölgesine ekleme (RL Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 1 iken karşılaşılır, sağ ve sol çift döndürme ile çözülür.
- B bölgesine ekleme (LR Imbalance): P'nin denge faktörü 2, L'nin denge faktörü -1 iken karşılaşılır, sol ve sağ çift döndürme ile çözülür.



Şekil 3.10'da bir AVL ağacına 2 değerini içeren yeni bir düğüm eklenmiştir. Oluşan yeni ağacın denge faktörleri hesaplanmış ve ağacın dengesiz olduğu görülmüştür. 6 değerini içeren düğümün denge faktörü 2 olduğu için, bu düğüm pivot olarak belirlenmiştir. P'nin denge faktörü 2, L'nin denge faktörü 1 olduğu ağaçta LL Imbalance vardır. Den gesizliğinin çözümü için pivot etrafında sağa doğru tek döndürme işlemi uygulanır. Elde edilen son ağaç dengededir ve AVL ağacı veri yapısı özelliklerini taşımaktadır.

Sekil 3.11

AVL Ağacına Düğüm Ekleme (RL Imbalance) ve Döndürme Örneği.



Şekil 3.11'de bir AVL ağacına 18 değerini içeren yeni bir düğüm eklenmiştir. Oluşan yeni ağacın denge faktörleri hesaplanmış ve ağacın dengesiz olduğu görülmüştür. 10 değerini içeren düğümün denge faktörü -2 olduğu için, bu düğüm pivot olarak belirlenmiştir. P'nin denge faktörü -2, R'nin denge faktörü 1 olduğu için ağaçta RL Imbalance vardır. Dengesizliğin çözümü için önce 20 değerini içeren düğüm etrafında sağa döndürme, sonrasında pivot etrafında sola döndürme işlemi uygulanır. Elde edilen son ağaç dengededir ve AVL ağacı veri yapısı özelliklerini taşımaktadır.

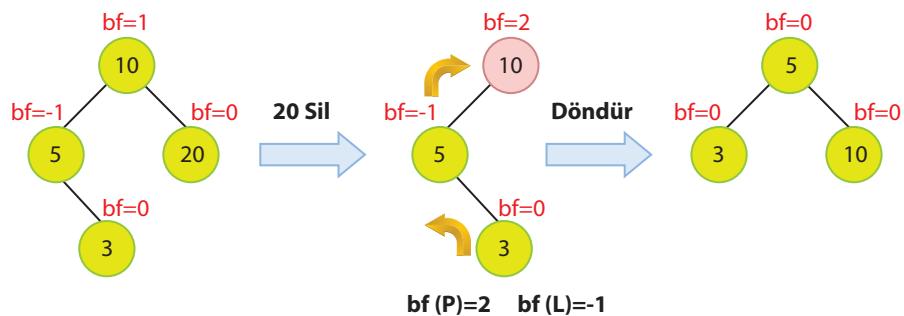
AVL Ağacından Düğüm Çıkarma ve Denge Korunumu

AVL ağacından düğüm çıkarılırken, ikili arama ağacındaki çıkış yöntemi izlenir. Düğüm çıkarıldıkten sonra oluşan yeni ağaç üzerinde denge faktörleri hesaplanır. Ağaçta bir dengesizlik olması durumunda dengesizliğin olduğu düğüm (pivot) tarafında bir veya iki tane döndürme işlemi uygulanır.

- Cıkarma işleminden sonra oluşan dengesizlik, dört ayrı şekilde görülebilir:
- A bölgesinden çıkışma (LL Imbalance): P'nin denge faktörü 2, L'nin denge faktörü 0 veya 1 iken karşılaşılır, pivot etrafında sağa doğru tek döndürme ile çözülür.
 - D bölgesinden çıkışma (RR Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 0 veya -1 iken karşılaşılır, pivot etrafında sola doğru tek döndürme ile çözülür.
 - C bölgesinden çıkışma (RL Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 1 iken karşılaşılır, sağ ve sol çift döndürme ile çözülür.
 - B bölgesinden çıkışma (LR Imbalance): P'nin denge faktörü 2, L'nin denge faktörü -1 iken karşılaşılır, sol ve sağ çift döndürme ile çözülür.

Sekil 3.12

AVL Ağacından Düğüm Çıkarma (LR Imbalance) ve Döndürme Örneği.



Şekil 3.12'de bir AVL ağacından 20 değerini içeren düğüm çıkarılmıştır. Oluşan yeni ağaçın denge faktörleri hesaplanmış ve ağaçın dengesiz olduğu görülmüştür. 10 değerini içeren düğümün denge faktörü 2 olduğu için, bu düğüm pivot olarak belirlenmiştir. P'nin denge faktörü 2, L'nin denge faktörü -1 olduğu için ağaçta LR Imbalance vardır. Denge-sizliğin çözümü için önce 3 değerini içeren düğüm etrafında sola döndürme, sonrasında pivot etrafında sağa döndürme işlemi uygulanır. Elde edilen son ağaç dengedededir ve AVL ağaçları veri yapısı özelliklerini taşımaktadır.

Verilen örnekleri bilgisayar ortamında bizzat denemeniz, ağaçlar hakkında deneyiminizin artmasına katkıda bulunacaktır.



DİKKAT

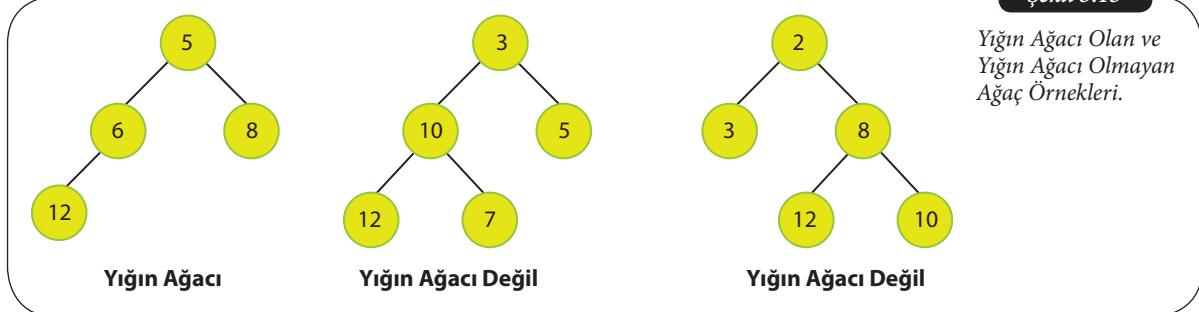
YİĞİN AĞAÇLARI

Yiğin ağaçları, bir veri kümesi içerisinde en küçük elemanın hızlıca bulunmasını sağlayan bir veri yapısıdır. Bu veri yapısında en küçük elemanı bulma, en küçük elemanı silme ve ağaçca eleman ekleme işlemleri hızlıca yapılabilir.

Aşağıda verilen iki özelliği sağlayan bir ikili ağaç, yiğin ağaçları veri yapısı olarak sınıflandırılır:

1. Ağaç bütünlüğü: Ağaçın son düzeyi hariç tüm düzeyleri, içerdikleri düğümler bakımından eksiksiz olmalıdır. Ağaçın son düzeyindeki düğümler de soldan sağa doğru doldu olmalıdır.
2. Heap özelliği: Bir düğümün sahip olduğu değer, düğümün çocuklarına ait değerlerden küçük veya eşit olmalıdır.

Şekil 3.13



Şekil 3.13'te üç adet ikili ağaç gösterilmiştir. Verilen örnekleri ayrı ayrı inceleyelim:

- Şekilde verilen ilk ağaçta son düzey hariç tüm düzeyler doludur, son düzey ise soldan sağa doludur. Tüm düğümlerdeki değerler, çocuklarına ait değerlerden küçüktür. Bu örnekte ağaç bütünlüğü ve heap özelliği sağlandığı için, bu ağaç bir yiğin ağaçıdır.
- Şekilde verilen ikinci ağaçta ağaç bütünlüğü sağlanmasına rağmen, heap özelliği sağlanmamaktadır. 10 değerine sahip düğümün 7 değerine sahip çocuğu olduğu için, bu ağaç bir yiğin ağaç değildir.
- Şekilde gösterilen üçüncü ağaçta son düzeydeki düğümlerin soldan sağa doğru doldu olmadığı görülmektedir. Dolayısıyla bu ağaç için bir bütünlük söz konusu değildir. Ağaç bütünlüğü sağlanmadığı için, bu ağaç bir yiğin ağaçıdır.

Bu konuda anlatılan yiğin ağaçları, minimum yiğin ağaçlarını kapsamaktadır. Yiğin ağaclarında en büyük elemanı baz alarak da işlem yapılabılır. Ağacın en üst noktasında en büyük elemanın yer aldığı, heap özelliğinde ise büyük olma durumunun arandığı yiğin ağaçları maksimum yiğin ağaçları olarak adlandırılır.



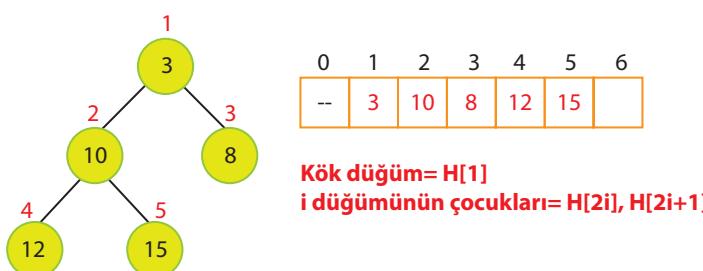
DİKKAT

Dizi ile Yiğin Ağacı Uygulaması

Yığın ağaçları, ağaç bütünlüğüne sahip ikili ağaçlar olduğu için, yiğin ağacı veri yapısını bir dizi kullanılarak programlanabilir. Bu sayede göstericilerden ve bağlı liste kullanımından kaçınılmış olunur.

Şekil 3.14

Yığın Ağacının Bir Dizi ile Gösterimi.



Şekil 3.14'te yiğin ağacının bir dizi ile ifade ediliş biçimini gösterilmiştir. N uzunlığında bir dizinin 1 numaralı indisine yiğin ağacının kökü yerleştirilir. Ağacın geri düğümleri de düzey içerisinde soldan sağa sıralanacak şekilde dizinin geri kalan indislerine yerleştirilir. Böylelikle i numaralı indiste yer alan düğümün çocukları, $2i$ ve $2i+1$ numaralı indislere denk gelmiş olur.

Yığın Ağacında En Küçük Elemanı Elde Etme

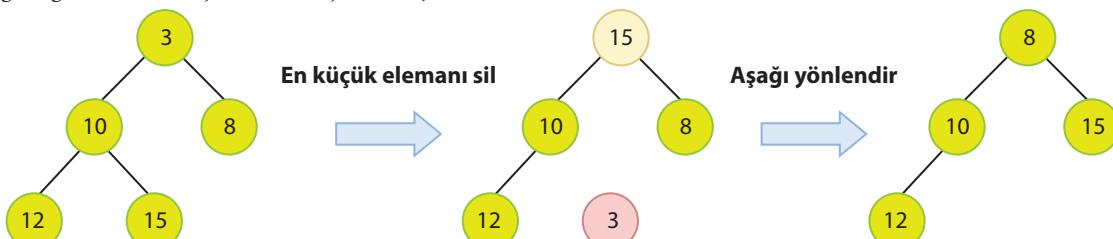
Yığın ağacının en küçük elemanı, ağacın kökünde yer almaktadır. Bu elemanı elde etmek için dizinin 1 numaralı indisine erişmek yeterlidir. N uzunlığunda H isimli bir dizi ile ifade edilen yiğin ağacında en küçük elemanı elde etmek için dizinin $H[1]$ elemanına erişmek yeterlidir.

Yığın Ağacından En Küçük Elemanı Çıkarma: Aşağı Yönlendirme

Yığın ağacının en küçük elemanı ağaçtan çıkarılırken, ağacın son düzeyinin en sağındaki düğüm ile ağacın kökü yer değiştirilir. Yer değiştirme işleminden sonra en küçük elemanın içeren ağacın son düzeyindeki en sağ düğüm ağaçtan çıkarılır. Ağacın heap özelliğini korumak için, ağacın kökü üzerine aşağı yönlendirme işlemi uygulanır. Bu düğüm, ağac içerisinde doğru yere gelinceye kadar çocukları ile karşılaşır ve değer olarak en küçük çocuk ile yer değiştirilir. Yiğin ağacından en küçük elemanı çalışma işleminin gösterildiği bir örnek, Şekil 3.15'te verilmiştir.

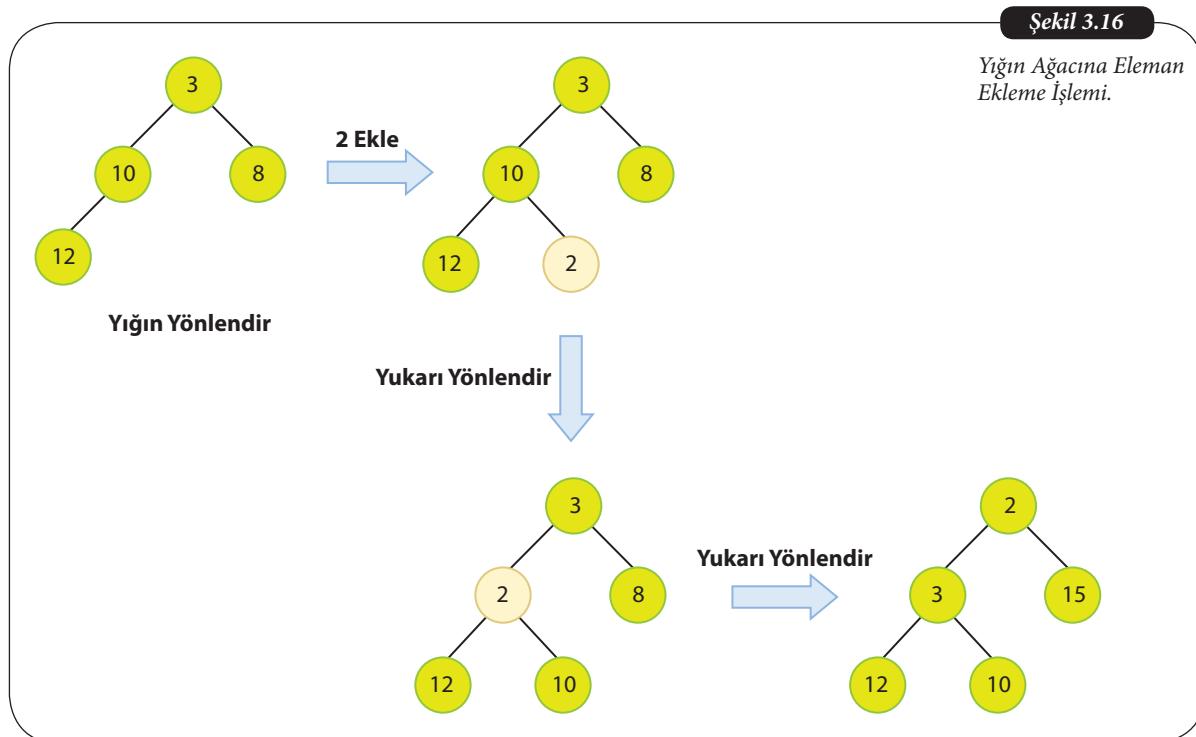
Şekil 3.15

Yığın Ağacından En Küçük Elemanı Çıkarma İşlemi.



Yığın Ağaçına Eleman Ekleme: Yukarı Yönlendirme

Yığın ağaçına yeni bir eleman eklerken, ağaçın son düzeyinin en sağında yeni bir düğüm yaratılır ve eklenecek değer bu yeni düğüme atanır. Oluşan yeni ağaç için ağaç bütünlüğünü korunmakta, fakat ağaçın heap özelliğini kaybolmaktadır. Ağaçın heap özelliğini sağlaması amacıyla, eklenen düğüm üzerinden yukarı yönlendirme işlemi yapılır. Eklenen yeni düğüm, ağaç içerisinde doğru yere gelinceye kadar ebeveyniyle karşılaşılır ve düğümün değeri ebeveynin değerinden küçük ise düğüm ile ebeveynin yeri değiştirilir. Şekil 3.16'da, yığın ağaçına eleman ekleme işleminin gösterildiği bir örnek verilmiştir.



ÖZTELME (HASH) TABLOLARI

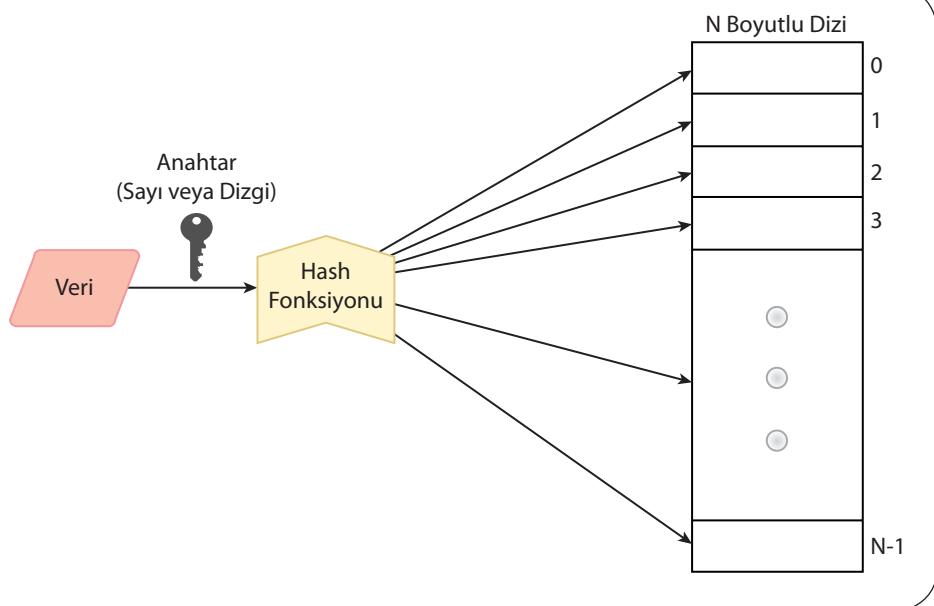
Öztleme tabloları ekleme, silme ve arama işlemlerinin çok hızlı bir şekilde yapılmasını sağlayan, verileri bir anahtar ve veri çifti şeklinde saklayan veri yapısıdır.

Öztleme tablolarındaki genel çalışma mantığı, verileri N boyutlu bir dizide tutmak ve verilere erişim için sayı veya dizgiden oluşan anahtarları kullanmaktadır. Öztleme tablosunda saklanacak bir veri için **hash fonksiyonuna** anahtar değeri gönderilir, fonksiyonun hesapladığı değer, verinin dizide tutulacağı indis olur. Öztleme tablolarının çalışma mantığı Şekil 3.17'de gösterilmiştir.

Hash Fonksiyonu: Öztleme tablolarında verilen bir anahtar için tablodaki indis değerini hesaplayıp döndüren fonksiyondur.

Sekil 3.17

Özetleme Tablolarının Çalışma Mantığı.

**ÖRNEK 3.6**

Anahtar olarak sayı değeri alan bir hash fonksiyonu.

```
int hash(int key, int N) {
    return key % N;
}
```

Örnek 3.6'da, anahtar olarak sayı değeri alan bir hash fonksiyonu gösterilmiştir. Bu fonksiyon, özetleme tablosu indisi olarak anahtar sayının dizi uzunluğuna modüler bölümünü döndürür.

ÖRNEK 3.7

Anahtar olarak dizgi değeri alan bir hash fonksiyonu.

```
int hash(char key[], int M, int N) {
    int i, sum = 0;

    for(i=0; i<M; i++) {
        sum += key[i];
    }

    return sum % N;
}
```

Örnek 3.7'de anahtar olarak dizgi değeri alan bir hash fonksiyonu gösterilmiştir. Bu fonksiyon, dizgideki her karakterin numerik değerini toplar. Elde edilen toplamın dizi uzunluğuna modüler bölümünü özetleme tablosu indisi olarak döndürür.

Çatışmalar

Özetleme tablolarında kullanılan hash fonksiyonları belirli bir algoritma sahiptir. Fonksiyon için tanımlı algoritma, fonksiyona verilen anahtar değeri doğrultusunda bir indis değerler hesaplar ve döndürür.

Hash fonksiyonu için tanımlanan algoritma, her anahtar değeri için farklı bir indis üretmeyebilir. Özetleme tablosunda fonksiyonun döndürdüğü indis değerine karşılık gelen alan dolu ise çatışma adı verilen durum ortaya çıkar.

Bir çalışma senaryosu gözlemlemek için Örnek 3.6'da verilen hash fonksiyonunu inceleyelim. Özetteleme tablomuzdaki dizi için N değerini 10 olarak kabul edelim.

- Anahtar=19, Veri=A olan ekleme işleminde A verisi 9 ($19 \% 10$) numaralı indise yerleştirilir.
- Anahtar=21, Veri=B olan ekleme işleminde B verisi 1 ($21 \% 10$) numaralı indise yerleştirilir.
- Anahtar=10, Veri=C olan ekleme işleminde C verisi 0 ($10 \% 10$) numaralı indise yerleştirilir.
- Anahtar=31, Veri=D olan ekleme işleminde D verisi 1 ($31 \% 10$) numaralı indise yerleştirilemez. 1 numaralı indiste B verisi bulunmaktadır, çalışma meydana gelmiştir.

Çalışmalar, hash fonksiyonları için istenmeyen durumlardır. Verimli ve etkin bir hash fonksiyonu, aşağıdaki özellikleri sağlamalıdır:

- i. Fonksiyon içerisinde hesaplamalar hızlı yapılmalıdır.
- ii. Hesaplama sonucu üretilen değerlerde minimum çalışma olmalıdır.
- iii. Özetteleme tablosundaki tüm alanlar kullanılabilir olmalıdır.
- iv. Özetteleme tablosundaki doluluklarda eşit dağılım sağlanmalıdır.

Çalışma Çözüm Yöntemleri

Bir hash fonksiyonu için çalışma oluşumu ihtimalini ortadan kaldırmak mümkün olmasa da çalışma ile karşılaşıldığında uygulanabilecek çözümler mevcuttur. Çalışma çözüm yöntemleri iki ana başlıkta incelenir:

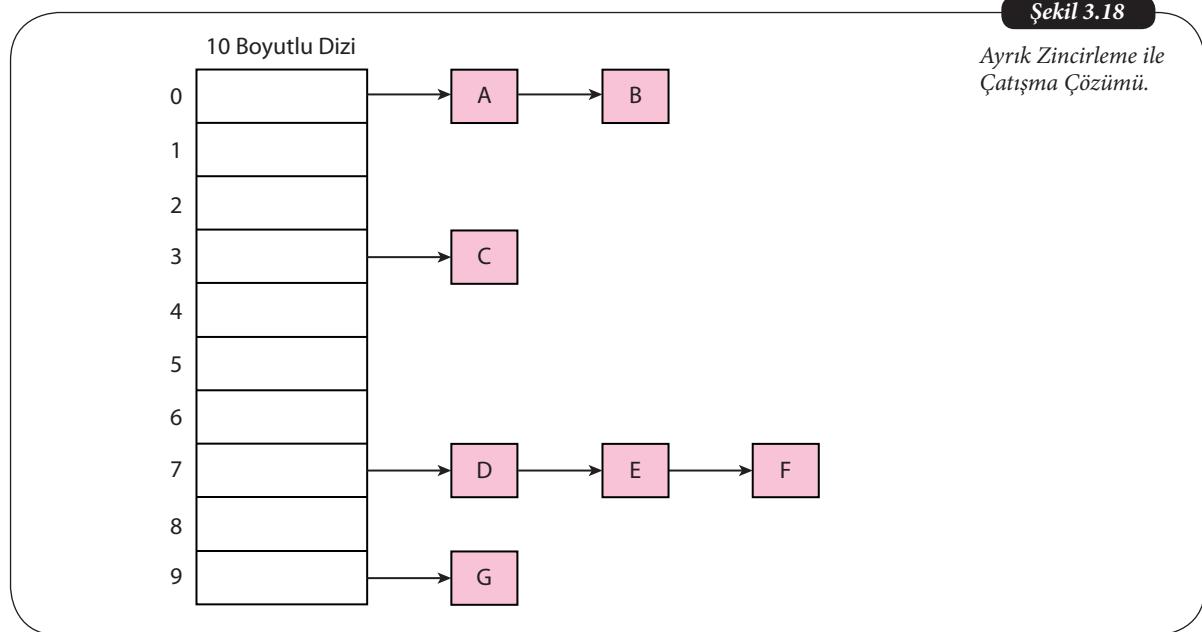
1. Ayrık Zincirleme (Separate Chaining)
2. Açık Adresleme (Open Addressing)

Ayrık Zincirleme (Separate Chaining)

Ayrık zincirleme yöntemiinde aynı indise karşılık gelen veriler, bir bağlı liste kullanarak saklanır. Böylelikle özetteleme tablosundaki bir alanda birden çok verinin saklanması sağlanır.

Şekil 3.18

Ayrık Zincirleme ile
Çalışma Çözümü.



Şekil 3.18'de ayrık zincirleme yöntemiyle çalışma çözümü için bir örnek gösterilmiştir. 10 boyutlu bir özetteleme tablosu için aşağıdaki varsayımlar yapılmıştır:

- $\text{hash}(\text{anahtar A}) = \text{hash}(\text{anahtar B}) = 0$
- $\text{hash}(\text{anahtar C}) = 3$
- $\text{hash}(\text{anahtar D}) = \text{hash}(\text{anahtar E}) = \text{hash}(\text{anahtar C}) = 7$
- $\text{hash}(\text{anahtar G}) = 9$

Listelenen varsayımlar doğrultusunda yaşanan çatışmaların çözümü ayrik zincirleme ile yapılmıştır. A ve B verileri ile D, E ve F verileri kendi aralarında birer bağlı listede tutulmuştur.

Açık Adresleme (Open Addressing)

Olası bir çalışma durumunda ikinci bir hash fonksiyonu kullanarak, tabloda boş bir alan aranan yönteme açık adresleme adı verilir. N boyutlu bir özetleme tablosunda, X anahtarı için açık adreslemenin genel formülü aşağıda gösterilmiştir:

$$h_i(X) = (\text{Hash}(X) + F(i)) \bmod N$$

Belirtilen formüldeki i değişkeni, anahtarın hash fonksiyonuna kaçinci kez gönderildiğini ifade eden sayıdır. F fonksiyonu çalışma çözümü için kullanılan ikinci fonksiyondur ve $F(0) = 0$ 'dır.

Açık adreslemede çalışma çözümü için kullanılan üç çeşit temel ikinci hash fonksiyonu bulunmaktadır:

- a. Doğrusal Sınama (Linear Probing): $F(i)=i$
- b. Karesel Sınama (Quadratic Probing): $F(i)=i^2$
- c. İkili Hash (Double Hashing): $F(i)=i * \text{Hash}_2(i)$

ÖRNEK 3.8

10 boyutlu bir özetleme tablosuna 18, 19, 20, 29, 30 anahtarlarına sahip verileri doğrusal sınama ile yerleştirme.

Anahtar	i	$F(i)$	$h_i(X) = (\text{Hash}(X) + F(i)) \bmod N$	Sonuç
18	0	$F(0) = 0$	$h_0(18) = (\text{Hash}(18) + F(0)) \bmod 10$	8
19	0	$F(0) = 0$	$h_0(19) = (\text{Hash}(19) + F(0)) \bmod 10$	9
20	0	$F(0) = 0$	$h_0(20) = (\text{Hash}(20) + F(0)) \bmod 10$	0
29	0	$F(0) = 0$	$h_0(29) = (\text{Hash}(29) + F(0)) \bmod 10$	9 (Çatışma)
29	1	$F(1) = 1$	$h_1(29) = (\text{Hash}(29) + F(1)) \bmod 10$	0 (Çatışma)
29	2	$F(2) = 2$	$h_2(29) = (\text{Hash}(29) + F(2)) \bmod 10$	1
30	0	$F(0) = 0$	$h_0(30) = (\text{Hash}(30) + F(0)) \bmod 10$	0 (Çatışma)
30	1	$F(1) = 1$	$h_1(30) = (\text{Hash}(30) + F(1)) \bmod 10$	1 (Çatışma)
30	2	$F(2) = 2$	$h_2(30) = (\text{Hash}(30) + F(2)) \bmod 10$	2

Örnek 3.8'de 10 boyutlu bir özetleme tablosuna doğrusal sınama ile veri eklenmiştir. 18, 19 ve 20 anahtarlarına sahip veriler ilk denemedede ($i=0$) sırasıyla 8, 9 ve 0 numaraları indislere eklenmiştir. 29 anahtarına sahip veri, ilk denemedede ($i=0$) 9 numaralı indise, ikinci denemedede ($i=1$) 0 numaralı indise eklenmeye çalışılmış fakat tablonun bu alanları dolu olduğu için çatışmalar meydana gelmiştir. 29 anahtarına sahip veri üçüncü denemedede ($i=2$) 1 numaralı indise eklenmiştir. 30 anahtarına sahip veri ilk denemedede ($i=0$) 0 numaralı indise, ikinci denemedede ($i=1$) 1 numaralı indise eklenmeye çalışılmış fakat tablonun bu alanları dolu olduğu için çatışmalar meydana gelmiştir. 30 anahtarına sahip veri üçüncü denemedede ($i=2$) 2 numaralı indise eklenmiştir.

SIRA SİZDE



3

Örnek 3.8'de gösterilen özetleme tablosuna veri ekleme işlemini karesel sınama fonksiyonunu kullanarak gerçekleştiriniz. Ekleme işlemlerindeki adımları ve sonuçları bir tablo içerisinde gösteriniz.

Özet



Ağaç veri yapısında yer alan temel kavramları, ikili ağaçları ve ikili arama ağaçlarını tanımlamak

Ağaç veri yapısı, verilerin birbirlerine temsili bir ağaç oluşturacak şekilde bağılandığı hiyerarşik bir veri modelidir. Düğümlerden ve düğümleri birbirine bağlayan dallardan meydana gelen bu veri yapısında yer alan diğer temel kavramlar kök, yol, yol uzunluğu, ebeveyn, çocuk, ağaç yüksekliği, düğüm yüksekliği ve düğüm derinliğidir.

İkili ağaçlar, her bir düğümün en fazla 2 çocuğa sahip olabildiği ağaç türüdür. Bu veri yapısında ekleme, silme ve arama işlemleri hızlı bir şekilde yapılmaktadır. İkili arama ağaçları, ikili ağaçların özel bir türüdür. Bu veri yapısında, ikili ağaç özelliklerine ek olarak düğümlerde yer alan veriler arasında büyülüklük-küçüklik ilişkisi bulunmaktadır. İkili ağaç özellikleri taşıyan bir ağaçın ikili arama ağaç olabilmesi için ağaçtaki her düğümün, sol alt ağacındaki tüm değerlerden büyük olması, sağ alt ağacındaki tüm değerlerden küçük veya eşit olması gerekmektedir.



Ağaçlar üzerinde gezinme yöntemlerini sıralamak

Ağaç üzerinde gezinme, bir ağaçın düğümlerini belirli bir algoritma ve sıra çerçevesinde dolaşma anlamına gelir. İkili ağaçlarda gezinme yapılrken özyinelemeli fonksiyonlardan faydalananır ve bu fonksiyonların kullanıldığı üç temel yöntem bulunmaktadır:

- Preorder Gezinme (Kök başta): Bu yöntemde önce kök, daha sonrasında sol alt ağaç, en son olarak da sağ alt ağaç üzerinde gezinme yapılır.
- Inorder Gezinme (Kök ortada): Bu yöntemde önce sol alt ağaç, daha sonrasında kök, en son olarak da sağ alt ağaç üzerinde gezinme yapılır.
- Postorder Gezinme (Kök sonda): Bu yöntemde önce sol alt ağaç, daha sonrasında sağ alt ağaç, en son olarak da kök üzerinde gezinme yapılır.



AVL ağaçları veri yapısını tanımlamak ve ağaç dengesinin korunması için gerekli döndürme işlemlerini açıklamak

AVL (Adelson – Velsky – Landis) ağaçları, ikili arama ağaçlarının özel bir türüdür. Bu veri yapısında ağaç içeresindeki denge korunmakta, sol alt ağaç ile sağ alt ağaç arasındaki yükseklik farkı en fazla 1 olabilmektedir.

AVL ağaçlarındaki düğümler için denge faktörü sol alt ağaçın yüksekliği ile sağ alt ağaçın yüksekliği arasındaki farka eşittir ve dengeli bir ağaç için denge faktörleri yalnızca -1, 0 ve 1 olabilir.

AVL ağaç üzerinde yapılan düğüm ekleme ve çıkarma işlemleri ağaçın dengesini bozabilir. Ağaçın dengesinin bozulduğu durumlarda, çeşitli döndürmeler aracılığıyla ağaç dengesi yeniden kurulur.

Ekleme ve çıkarma işleminden sonra oluşan dengesizlik, dört ayrı şekilde görülebilir:

- A bölgesinde ekleme/çıkarma (LL Imbalance): P'nin denge faktörü 2, L'nin denge faktörü 0 veya 1 iken karşılaşılır, pivot etrafında sağa doğru tek döndürme ile çözülür.
- D bölgesinde ekleme/çıkarma (RR Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 0 veya -1 iken karşılaşılır, pivot etrafında sola doğru tek döndürme ile çözülür.
- C bölgesinde ekleme/çıkarma (RL Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 1 iken karşılaşılır, sağ ve sol çift döndürme ile çözülür.
- B bölgesinde ekleme/çıkarma (LR Imbalance): P'nin denge faktörü 2, L'nin denge faktörü -1 iken karşılaşılır, sol ve sağ çift döndürme ile çözülür.



Yığın ağaçlarının temel özelliklerini ve bu veri yapısında ekleme/çıkarma işlemlerini özetlemek

Yığın ağaçları, bir veri kümesi içerisinde en küçük elemanın hızlıca bulunmasını sağlayan bir veri yapısıdır. Bir ikili ağaçın yığın ağaç olarak sınıflandırılabilmesi için aşağıda verilen iki özelliği sağlaması gereklidir:

- Ağaç bütünlüğü: Ağaçın son düzeyi hariç tüm düzeyleri, içerdikleri düğümler bakımından eksiksiz olmalıdır. Ağaçın son düzeyindeki düğümler de soldan sağa doğru dolu olmalıdır.
- Heap özelliği: Bir düğümün sahip olduğu değer, düğümün çocuklarına ait değerlerden küçük veya eşit olmalıdır.

Ağaç bütünlüğe sahip ikili ağaçlar olan yığın ağaçları, bir dizi kullanarak programlanabilir. Ağaçın kökünde yer alan ağaçın en küçük elemanını elde etmek için dizinin 1 numaralı indisine erişmek yeterlidir.

Yığın ağaçının en küçük elemanı ağaçtan çıkarılırken, ağaçın son düzeyinin en sağındaki düğüm ile ağaçın kökü yer değiştirilir. Yer değiştirme işleminden sonra en küçük elemanı içeren ağaçın son düzeyindeki en sağ düğüm ağaçtan çıkarılır. Ağaçın heap özelliğini korumak için, ağaçın kökü üzerine aşağı yönlendirme işlemi uygulanır.

Yığın ağaçına yeni bir eleman eklerken, ağaçın son düzeyinin en sağında yeni bir düğüm yaratılır ve ekle-

necek değer bu yeni düğüme atanır. Ağacın heap özelliğini sağlamak için eklenen düğüm üzerinden yukarı yönlendirme işlemi yapılır.



Özetleme tablosu veri yapısını ve özetleme fonksiyonlarını tanımlamak

Özetleme tablosu, verileri bir anahtar ve veri çifti şeklinde saklayan özel bir veri yapısıdır. Bu veri yapısında ekleme, silme, arama gibi işlemler çok hızlı bir şekilde yapılabilir.

Özetleme tablolarındaki genel çalışma mantığı, verileri N boyutlu bir dizide tutmak ve verilere erişim için sayı veya dizgiden oluşan anahtarı kullanmaktadır. Özetleme tablosunda saklanacak bir veri için hash fonksiyonuna anahtar değeri gönderilir, fonksiyonun hesapladığı değer, verinin dizide tutulacağı indis olur.



Özetleme fonksiyonlarında meydana gelen çatışmaları çözümleme yöntemlerini listelemek

Hash fonksiyonu için tanımlanan algoritma, her anahtar değeri için farklı bir indis üretmeyebilir. Özetleme tablosunda fonksiyonun döndürdüğü indis değerine karşılık gelen alan dolu ise çatışma adı verilen durum ortaya çıkar.

Bir hash fonksiyonu için çatışma oluşumu ihtimalini ortadan kaldırmak mümkün olmasa da ayrık zincirleme ve açık adresleme olarak bilinen iki yöntemle çatışmalara karşı aksiyon alınabilir.

Ayrık zincirleme yönteminde aynı indise karşılık gelen veriler, bir bağlı liste ile saklanır. Böylelikle özetleme tablosundaki bir alanda birden fazla veri saklanabilir. Açık adresleme yönteminde çatışmalar için ikinci bir hash fonksiyonu kullanılarak, tabloda boş alan aranır. Kullanılan ikinci fonksiyon doğrusal sinama, karesel sinama veya ikili hash olabilir.

Kendimizi Sınayalım

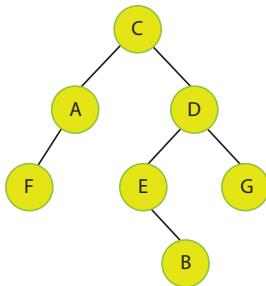
1. Ağaç veri yapısında yer alan düğümleri birbirine bağlayan kenarlara ne ad verilir?

- a. Çocuk
- b. Ebeveyn
- c. Dal
- d. Kök
- e. Ağaç yüksekliği

2. Aşağıdakilerden hangisi, sayısal değerler saklayan bir ikili arama ağacının Inorder gezinme yöntemi ile dolaşımıyla elde edilebilecek bir sayı dizilimi **olamaz**?

- a. 1 2 3 4 5
- b. 1 3 5 7 9
- c. 2 3 6 8 9
- d. 2 4 5 7 9
- e. 3 5 4 2 8

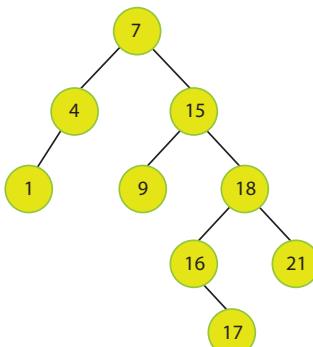
3.



Yukarıda gösterilen ikili ağaç, Preorder gezinme yöntemi ile dolaşıldığında hangi harf dizilimi elde edilir?

- a. C A F D E B G
- b. C A D F E G B
- c. F A C E B D G
- d. F A B E G D C
- e. A B C D E F G

4.



Yukarıdaki şekilde gösterilen ikili arama ağacının yüksekliği kaçtır?

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

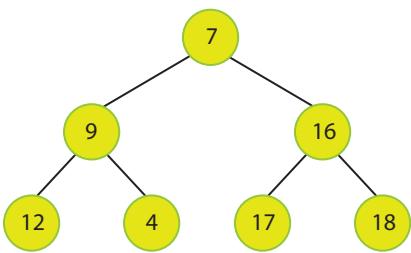
5. AVL ağaçları ile ilgili aşağıda verilen ifadelerden hangisi doğru **değildir**?

- a. AVL ağaçları, ikili arama ağaçlarının özel bir türüdür.
- b. Bir AVL ağacının kökünde yer alan düğümü silmek mümkün değildir.
- c. Bir AVL ağaçları her zaman denge olmalıdır.
- d. Ağaca eklenen her düğüm sonrası bir döndürme yapılır.
- e. Ağaç düğümlerinin denge faktörü -1, 0 veya 1 olabilir.

6. Aşağıdaki ifadelerden hangisi yığın ağaçları için geçerlidir?

- a. Her ikili arama ağaçları, aynı zamanda bir yığın ağaçıdır.
- b. Yığın ağaçları veri yapısı oluşturmada bir diziden faydalananlamaz.
- c. Ağaca eleman ekledikten sonra aşağı yönlendirme işlemi yapılır.
- d. Ağacın en küçük elemanını sildikten sonra yukarı yönlendirme işlemi yapılır.
- e. Yığın ağaçında sağ çocuğu olup, sol çocuğu olmayan bir düğüm olamaz.

7.



Yukarıda gösterilen ağaç üzerinde aşağıdaki işlemlerden hangisi uygulandığında, yiğin ağacının heap özelliği sağlanmış olur?

- 5 değeri içeren yeni bir düğüm eklemek
 - 4 değerini içeren düğümün değerini 10 yapmak
 - 9 değerini içeren düğümün değerini 6 yapmak
 - 18 değerini içeren düğümü silmek
 - 17 değerini içeren düğümü silmek
- 8.** Geliştirdiği programda verileri bir anahtar ve veri çifti olarak saklamak, bu verilere anahtarlar aracılığıyla doğrudan erişmek isteyen bir programcı, aşağıdaki veri yapılarından hangisini kullanmalıdır?
- Dizi
 - Bağılı liste
 - Özetleme tablosu
 - İkili arama ağaçısı
 - Kuyruk

9. Hash fonksiyonlarında yaşanan çatışmaları çözerken, bağlı liste kullanımı ile aynı indiste farklı verilerin tutulmasını sağlayan yöntem hangisidir?

- Ayrık zincirleme
- Doğrusal sınaması
- Açık adresleme
- Karesel sınaması
- İkili hash

10. Aşağıdakilerden hangisi etkin bir hash fonksiyonundan beklenen bir özellik **değildir**?

- Minimum sayıda çalışma oluşturmak
- Kısa süre içerisinde cevap döndürmek
- Tabloda kullanılan alanlarda eşit dağılım sağlamak
- Tablodaki tüm alanların kullanılabilir olmasını sağlamak
- Olası bir çatışmada programı durdurmak

Kendimizi Sınavalım Yanıt Anahtarı

1. c Yanınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
2. e Yanınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
3. a Yanınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
4. d Yanınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
5. d Yanınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
6. e Yanınız yanlış ise “Yıgın Ağaçları” konusunu yeniden gözden geçiriniz.
7. b Yanınız yanlış ise “Yıgın Ağaçları” konusunu yeniden gözden geçiriniz.
8. c Yanınız yanlış ise “Öztleme Tabloları” konusunu yeniden gözden geçiriniz.
9. a Yanınız yanlış ise “Öztleme Tabloları” konusunu yeniden gözden geçiriniz.
10. e Yanınız yanlış ise “Öztleme Tabloları” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

Aşağıda verilen program, ikili ağaçlara düğüm ekleme ve ağaç düğümleri üzerinde Preorder, Inorder ve Postorder gezinme yöntemlerini içermektedir. Ana fonksiyon içerisinde dört düğüme sahip bir ağaç oluşturulmuş ve bu ağaç tüm gezinme yöntemleri ile dolaşılıp, düğüm değerleri ekrana yazdırılmıştır.

```
#include<stdio.h>

typedef struct TreeNode {
    int data;
    struct TreeNode* left;
    struct TreeNode* right;
} TreeNode;

TreeNode* createNode(int x) {
    TreeNode* t = (TreeNode*) malloc(sizeof(TreeNode));
    t->data = x;
    t->left = t->right = NULL;
    return t;
}

void preorder(TreeNode *root) {
    if(root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

void inorder(TreeNode *root) {
    if(root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

void postorder(TreeNode *root) {
    if(root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

int main(void) {
    TreeNode *root = createNode(7);
    root->left = createNode(8);
    root->right = createNode(9);
    root->left->left = createNode(2);

    printf("\nPreorder gezinme: ");
    preorder(root);
    printf("\nInorder gezinme: ");
    inorder(root);
    printf("\nPostorder gezinme: ");
    postorder(root);

    getch();
    return 0;
}
```

Sıra Sizde 2

Aşağıda gösterilen program ikili arama ağacına düğüm ekleme ve bu ağaçtan düğüm çıkarma fonksiyonlarını içermektedir. Bu programda tanımlı fonksiyonları kullanarak, ana fonksiyon içerisinde ikili arama ağacı oluşturulabilir, oluşturulan ağaçta düğüm eklenebilir ve ağaçtan düğüm silinebilir. Programa eklenecek bir gezinme yöntemi ile ağacın son halindeki düğümler ekranaya yazdırılabilir.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct TreeNode {
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
} TreeNode;

TreeNode* findMin(TreeNode *node) {
    if(node == NULL)
        return NULL;
    if(node->left)
        return findMin(node->left);
    else
        return node;
}

TreeNode* insertNode(TreeNode *node, int x) {
    if(node == NULL) {
        TreeNode *t = (TreeNode *) malloc(sizeof(TreeNode));
        t->data = x;
        t->left = t->right = NULL;
        return t;
    }
    else if(x > node->data) {
        node->right = insertNode(node->right, x);
    }
    else {
        node->left = insertNode(node->left, x);
    }
}
}

TreeNode* deleteNode(TreeNode *node, int x) {
    TreeNode *temp;

    if(node == NULL) {
        printf("Silinecek düğüm bulunamadı.\n");
    }
    else if(x < node->data) {
        node->left = deleteNode(node->left, x);
    }
    else if(x > node->data) {
        node->right = deleteNode(node->right, x);
    }
    else {
        if(node->right && node->left) {
            temp = findMin(node->right);
            node->data = temp->data;
            node->right = deleteNode(node->right, temp->data);
        }
        else {
            temp = node;
            if(node->left == NULL)
                node = node->right;
            else if(node->right == NULL)
                node = node->left;

            free(temp);
        }
    }
}

return node;
}
```

Sıra Sizde 3

10 boyutlu bir özetleme tablosuna 18, 19, 20, 29, 30 anahtarlarına sahip verileri karesel sınamla ile yerleştirelim.

Anahtar	i	F(i)	$h_i(X) = (\text{Hash}(X) + F(i)) \bmod N$	Sonuç
18	0	$F(0) = 0$	$h_0(18) = (\text{Hash}(18) + F(0)) \bmod 10$	8
19	0	$F(0) = 0$	$h_0(19) = (\text{Hash}(19) + F(0)) \bmod 10$	9
20	0	$F(0) = 0$	$h_0(20) = (\text{Hash}(20) + F(0)) \bmod 10$	0
29	0	$F(0) = 0$	$h_0(29) = (\text{Hash}(29) + F(0)) \bmod 10$	9 (Çatışma)
29	1	$F(1) = 1$	$h_1(29) = (\text{Hash}(29) + F(1)) \bmod 10$	0 (Çatışma)
29	2	$F(2) = 4$	$h_2(29) = (\text{Hash}(29) + F(2)) \bmod 10$	3
30	0	$F(0) = 0$	$h_0(30) = (\text{Hash}(30) + F(0)) \bmod 10$	0 (Çatışma)
30	1	$F(1) = 1$	$h_1(30) = (\text{Hash}(30) + F(1)) \bmod 10$	1

Yararlanılan ve Başvurulabilecek Kaynaklar

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

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

Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

4

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Algoritmaların hesaplamadaki rolünü açıklayabilecek,
 - 🕒 Algoritma ile problem çözmenin genel aşamalarını listeleyebilecek,
 - 🕒 Tekrarlamalı döngülerle algoritma tasarımını özetleyebilecek,
 - 🕒 Küçült-Fethet yöntemi ile algoritma tasarımını açıklayabilecek,
 - 🕒 Böl-Fethet yöntemi ile algoritma tasarımını açıklayabilecek
- bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- Algoritma Tasarımı
- Döngü Algoritmaları
- Küçült-Fethet (Decrease-Conquer) Yöntemi
- Böl-Fethet (Divide-Conquer) Yöntemi

İçindekiler

Algoritmalar ve Programlama

Algoritma Tasarımı

- 
- Giriş
 - ALGORİTMA İLE PROBLEM ÇÖZME
 - ALGORİTMA TASARLAMA TEKNİKLERİ

Algoritma Tasarımı

GİRİŞ

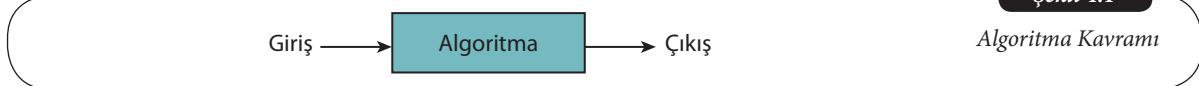
Algoritma belirli değerleri ya da değerler kümesini giriş olarak alan, istenilen amaca uygun olarak bu değerleri bir takım işlemlerden geçirerek bir çıktı ya da çıktı kümesi oluşturan işlemlerin bütünü olarak tanımlanabilir (T. H. Cormen, C. E. Leisers, R. L. Rivest, C. Stein (2009)). Bir hesaplama problemini algoritma ile çözebiliriz. Eğer giriş ve çıkış arasındaki ilişki matematiksel olarak belli ise bunu algoritmalar ile çözebiliriz. Temel ve pratikte sık karşılaşılan problemlerden birisi de sıralama problemidir. Sıralama problemini şu şekilde tanımlayabiliriz. Elimizde n tane sayı vardır ve bunları küçükten büyüğe doğru sıralamak istiyoruz. Örnek verecek olursak:

Giriş: 5, 6, 3, 7, 1, 0, 12, 15, 20, 30

Çıkış: 0, 1, 3, 5, 6, 7, 12, 15, 20, 30

Bu problem için tasarlanaçacak algoritma, bir sayı dizisini giriş olarak alacak ve işlemini tamamladıktan sonra bu sayıların sıralı hâlini çıkış verisi olarak geri döndürecektir. Sıralama problemi günlük hayatımızda sıkça karşılaştığımız bir problemdir. Örneğin, bir bölümdeki öğrencilerin sıralı listesini kullanmak istediğimizde, kayıtlı bütün öğrencileri numaralarına ya da isimlerine göre bir sıralama algoritması ile sıralamamız gerekmektedir. Algoritmaların işleyiş şeması Şekil 4.1'de görülmektedir.

Sekil 4.1



Algoritma Kavramı

Hangi Problemler Algoritmalar ile Çözülebilir?

Günlük hayatımızda birçok problem ile karşılaşmaktayız. Karşılaştığımız bu problemleri çözmek için algoritmaları kullanırız. Bir örnek verecek olursak İstanbul'dan Sivas'a araba ile gitmek istediğimizi varsayıyalım. Hangi yolları kullanırsak en kısa mesafede istediğimiz yere ulaşabiliriz? Günümüzde artık yol tarifleri için navigation cihazları kullanılmaktadır. Bu cihazlar ulaşmak istediğimiz yer için en kısa mesafeyi hesaplayıp yol tarifleri verebilmektedir. Bu cihazlarda, çeşitli ölçütler doğrultusunda en kısa mesafeyi hesaplayan algoritmalar çalışmaktadır.

Artık İnternette istediğimiz bilgiye kolaylıkla erişebiliyoruz. Bir arama motoruna ulaşmak istediğimiz bilgiyi yazdığımızda, ilgili bütün sayfaları ve dokümanları sıralayabilmektedir. Bu işlem esnasında ise sadece ilgili sayfaları gösterebilmek için bir metin arama

algoritması çalışmaktadır. Arama motorunun sonuç ekranında en çok ilgili olandan en az ilgili olana kadar bütün sonuçları listelemektedir. İnternetin yaygınlaşmasıyla arama motorları bilgiye ulaşmamızı kolaylaştırmaktadır.

Bilgisayarları yaşamımızın her alanında kullanmaya başladık. Örneğin, bankacılık işlemlerinde hem bilgisayarlarımızı hem de akıllı telefonlarımızı sıklıkla kullanmaya başladık. İşlemlerin çoğu elektronik olarak gerçekleştirildiği için bankacılık sistemlerinin güvenliği de önemini giderek arttırdı. Bankalar güvenliklerini artırmak için çeşitli şifreleme algoritmaları kullanmaktadır.

Akıllı telefonların kullanımı yaygınlaşıkça mobil uygulamaların önemi de artmaktadır. Akıllı telefonlarımızda kullandığımız uygulamalar ve oynadığımız oyunlar, algoritma kullanımının birer örneğidir. Akıllı telefonlarımızda ya da dijital fotoğraf makinelerimizde yüz tanıma algoritmaları kullanılmaktadır. Böylece resim çekerken yüz bölgesine odaklılama yapılmaktadır.

Teknolojik Olarak Algoritmalar

Algoritmalarla çalışırken kendi kendimize şu soruyu sorabiliriz: Bilgisayarlarımız sonsuz hızda sahip ve bilgisayar hafızaları ücretsiz olsaydı yine de algoritmalar üzerinde çalışmamız gereklidir mi? Bu soruya vereceğimiz cevap şüphesiz evet olurdu. Bulduğumuz çözüm yönteminin doğru cevap ile sonuçlandığını gösterebilme için de algoritmalarla çalışmamız gerekmektedir. Her ne kadar günümüzde bilgisayarlar oldukça hızlı olsa da sonsuz hızda değil ve ayrıca bilgisayar hafızalarının da belirli bir maliyeti bulunmaktadır. Başka bir ifadeyle, sonsuz hızda bir bilgisayarımız olmadığı için algoritmalar ve algoritmaların hızı üzerinde çalışmak neredeyse bir zorunluluktur. Şimdi, ünitemizin başında verdiğimiz sıralama algoritmasını düşünelim. Elimizdeki rastgele 100 tane sayıyı sıralamak istiyorsak günümüz bilgisayarlarında hangi algoritmayı kullandığımızın fazla bir önemi olmayacağıdır. Seçtiğimiz algoritma muhtemelen milisaniyeler içerisinde sonuç verecektir. Ancak, 100 yerine 10^{100} tane sayıyı sıralamak istersek bu sefer çok kısa sürelerde sonuç beklememiz söz konusu olamayacaktır. Bu sebeple, algoritmalar üzerinde çalışılması bir zorunluluk hâline gelmiştir. Algoritma tasarımını yaparken nerede ve ne şekilde kullanacağımızı düşünerek bilgisayarın hızını ve hafıza kapasitesini de göz önünde bulundurmamız gerekmektedir.

ALGORİTMA İLE PROBLEM ÇÖZME

Daha önce ifade edildiği üzere, algoritmaları problem çözmek için kullanmaktayız. Temel olarak bir algoritma tasarım ve analiz sürecinin işleyiş şeması Şekil 4.2'de verilmektedir. Bu şekilde görüldüğü üzere algoritma tasarımında ilk basamak problemi bütün ayrıntılarıyla anlamaktır. Problem tanımını okuyup kafamızda hiçbir şüphe kalmayacak şekilde anlamamız gerekmektedir. Algoritmayı tasarlamaya geçmeden önce ufak veri setleri üzerinde elle deneme yapmak, problemi anlamamızı kolaylaştıracaktır. Karşılaşabilecek bütün özel durumları da düşünmemiz gerekmektedir.

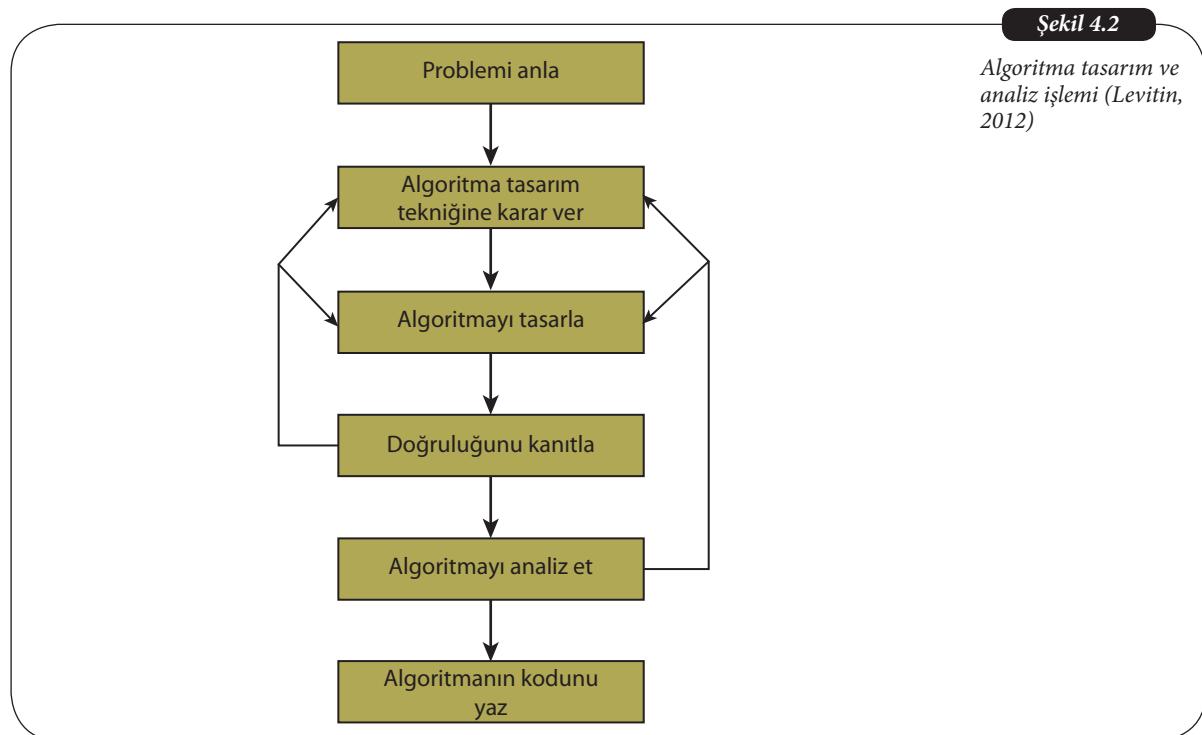
Sadece toplama ve çıkarma operasyonlarını kullanarak bölme işlemi yapan bir algoritma tasarlamak istediğimizi düşünelim. Böyle bir problemde karşılaşabileceğimiz özel durum sıfır bölme işlemi olabilir ve bölme işleminin sonucu tanımsız olacaktır.

Problemi anlamak, algoritma tasarlamanın birinci aşamasıdır.

SIRA SİZDE



Kitabımızın 2. ünitesinde anlatılan dizilere, bir sayı ekleme işlemi yapmak istediğimizde karşılaşabileceğimiz özel durum ne olabilir?



Problemi bütün ayrıntılarıyla anladıkten sonra düşünmemiz gereken husus, tasarılayacağımız algoritmayı hangi konfigürasyona sahip bilgisayarda çalıştırmak istedigimizdir. Kullanacağımız bilgisayarın işlemci hızı, hafıza kapasitesi vb. özellikleri tasarlayacağımız algoritmda etkilidir. Problemimiz çok karmaşık ve çözümü uzun süre gerektiriyorsa paralel programlama yöntemlerini kullanmaya karar verebiliriz. Problemimizi küçük parçalara böldükten sonra her bir parçayı ayrı bir bilgisayarda çözüp sonuçları birleştirerek asıl sonucu elde edebiliriz. Böylece problemimizi çok daha hızlı bir şekilde çözmüş oluruz.

Sonraki aşamada, algoritma tasarlama süreci gelmektedir. Algoritmaları tasarlarken diziler, kuyruklar, ağaçlar vb. veri yapılarından faydalananabiliriz. Veri yapısını seçerken mutlaka avantajlarını ve dezavantajlarını dikkate almamız gerekmektedir. Ayrıca kullanacağımız programlama dili de algoritma tasarımında değerlendirmemiz gereken hususlardan biri olmalıdır. Nesne tabanlı bir dil kullanacaksak, tasarımlarımız da bu dile uygun olmalıdır. Algoritma tasarımını yaparken doğal konuşma dili kullanabilirmiz. Ancak programcılar arasında bir bütünlük sağlamak ve problemin çözümünü daha iyi ifade edebilmek amacıyla sözde kod (pseudoocode) kullanabiliriz. Sözde kod, doğal dil ile programlama dili arasında bir problemin çözümünü ifade ediş biçimidir.

Kodlayacağımız algoritmayı tasarlarken, kodu çalıştıracağımız bilgisayarın konfigürasyonu çok önemlidir.

Üç tane sayının ortalamasını hesaplayan bir algoritma tasarlayıp sözde kod ile ifade edelim:

ÖRNEK 4.1

- 1) Sayıların değerlerini *sayı1*, *sayı2* ve *sayı3* olarak belirle
- 2) $ortalama = (\text{sayı1} + \text{sayı2} + \text{sayı3}) / 3$
- 3) Kullanıcıya *ortalama* değerini göster

İkinci dereceden bir denklemin köklerini bulan algoritmayı tasarlayınız ve bunu sözde kod ile ifade ediniz.



SIRA SİZDE

Tasarlanan algoritmanın doğru şekilde çalıştığı mutlaka teyit edilmelidir.

Algoritma tasarımını ve analizi sürecinin son adımı ilgili algoritmanın kodunu yazmaktadır.

Tasarladığımız algoritmayı göstermenin diğer bir yöntemi de akış diyagramı kullanmaktadır. Akış diyagramıyla basit algoritmaları rahatlıkla gösterebiliriz. Ancak karmaşık algoritmalar akış diyagramı ile gösterime çok uygun olmayabilir (Levitin, 2012).

Algoritmaları tasarlarken iteratif veya özyinelemeli fonksiyonlardan hangisini kullanacağımıza karar verip, tasarımını bu doğrultuda yapmamız gerekmektedir. Ünitemizin ilerleyen kısımlarında özyinelemeli fonksiyonlar hakkında detaylı açıklamalara yer verilmiştir.

Algoritmayı tasarladıktan sonra, algoritmamızın bütün girişler için doğru sonucu vereceğini doğrulamamız gerekmektedir. Bu işlem bazı algoritmalar için nispeten kolay olmakla birlikte bazı algoritmalar için karmaşık bir süreç gerektirmektedir. Bir algoritmanın hatalı çalıştığını gösterebilmek için sadece bir tane yanlış sonuç yeterlidir, ancak doğru çalıştığını ispatlamak için bütün verilerde doğru çalıştığını teyit etmek gereklidir. Örnek olarak sıralama algoritmasını ele alalım. Problemimizin kısıtlarını da düşünerek, tasarladığımız algoritmanın farklı eleman sayısı ve farklı elemanları sıralamak istediğimizde doğru sonucu vermesi gerekmektedir. Daha önce belirttiğimiz özel durumları da tasarımında göz önünde bulundurmaliyiz. Örneğin, bölme işlemi yapan algoritmanın, sıfır bölüm için tanımsız sonucunu vereceği de kontrol edilmelidir.

Hazırladığımız algoritmanın kodunu yazmaya başlamadan önceki son işlemimiz algoritmayı analiz etmektir. Algoritmanın çalışma zamanı, başka bir ifadeyle, algoritma karmaşıklık düzeyi ve hafıza gereksinimleri analiz edilmelidir. Algoritmaların nasıl analiz edildiği kitabımızın sonraki ünitesinde ayrıntılı şekilde ele alınmaktadır. Son işlemimiz, seçtiğimiz programlama dilinde algoritmamızın kodunu yazmaktır. Bu esnada, kullanacağımız programlama dilinin sunduğu imkânlar ölçüsünde bazı hazır fonksiyonlardan da faydalananmamız mümkün olabilir. Hazır fonksiyonlar beklenelerimizi tam olarak karşılamıyor ise kendi fonksiyonlarımızı kodlamamız gerekecektir. Kodu yazdıktan sonra belirli testlerden geçirmek, programın güvenilirliğini artıracaktır.

ALGORİTMA TASARLAMA TEKNİKLERİ

Algoritmaları temel olarak iki gruba ayırlabiliyoruz. Bunlar döngü (tekrarlama) algoritmaları ve özyinelemeli fonksiyon algoritmalarıdır. Döngü algoritmalarında, problemin çözümünü döngü içerisindeki tekrarlarla buluruz. Özyinelemeli fonksiyon algoritmalarında ise yazdığımız fonksiyon kendi içerisinde yine kendini çağırarak sonuca ulaşır. Her iki grubun da kendine has özelliklerini vardır. Bu bölümde, tekrarlama ve özyinelemeli fonksiyon algoritmalarının tasarım tekniklerini ele alacağız.

Döngü-Tekrarlama Algoritmaları

Elimizde n tane elemandan oluşan bir dizi olduğunu düşünelim. Bu dizinin içerisindeki en büyük elemanı nasıl buluruz? Bu problemde ilk akla gelen çözüm bütün elemanlara tek tek bakıp en büyük olan elemanı bulmak olabilir. Bu işlemi şu şekilde yapabilirimiz. Başlangıçta, dizinin ilk elemanını en büyük kabul ederiz. Daha sonra, en büyük kabul ettiğimiz değeri sırasıyla dizinin bütün elemanlarıyla karşılaşırız. Herhangi bir sıradaki elemanın değeri, en büyük kabul ettiğimiz değerden büyükse en büyük eleman değerimizi güncelleriz. Dizinin sonraki elemanlarını karşılaşırken güncellenmiş en büyük değerimizi kullanırız. Dizinin sonuna geldiğimizde ise dizinin tamamı açısından en büyük elemanı bulmuş oluruz. En büyük elemanı bulma algoritması için C dilinde yazılmış program örneği aşağıda verilmiştir:

```
#include <stdio.h>

// En büyük değeri bulan fonksiyon
int maksimum(int A[], int N)
{
    int i;
    int max;

    max = A[0];

    for (i = 1; i < N; i++)
    {
        if (max < A[i])
            max = A[i];
    }

    return max;
}

// Ana fonksiyon
int main()
{
    int a;
    int A[] = { -32, 13, -13, -6, 0, 8 };

    a = maksimum(A, 6);

    printf("Dizinin en büyük elemanı: %d\n", a);
}
```

Bir dizideki en küçük elemanı bulan fonksiyonu yazınız.



SIRA SİZDE

Tekrarlama algoritmaları için ikinci örneğimiz, verilen bir dizide elemanların toplamını hesaplayan fonksiyondur. Bu fonksiyon diziyi ve dizideki eleman sayısını giriş olarak almaktadır. Öncelik olarak toplam değerini tutacağımız değeri sıfır yapmaktadır. Daha sonra bir döngü içerisinde dizinin bütün elemanlarını sırasıyla bu toplam değerine ekliyoruz. Dizinin sonuna geldiğimizde elimizdeki toplam değerini, aradığımız sonucu oluşturmaktadır.

```
#include <stdio.h>

// Toplama fonksiyonu
int Toplama(int A[], int N) {
    int i;
    int toplam = 0;

    for (i = 0; i < N; i++) {
        toplam += A[i];
    }

    return toplam;
}

// Ana fonksiyon
int main()
{
    int a;
    int A[] = { -1, 13, 17, -6, 0, 18 };

    a = Toplama(A, 6);

    printf("Dizinin elemanlarının toplamı : %d\n", a);
}
```

Matematikte sıkça karşımıza çıkan ve bilinen problemlerden biri de Fibonacci serisi hakkındadır. Bilindiği üzere, bu serideki bir eleman kendinden önce gelen iki elemanın toplamına eşittir. Bu doğrultuda, Fibonacci serisi aşağıdaki şekilde ilerlemektedir:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Şimdi, bu serinin elemanlarını hesaplayan tekrarlamalı bir algoritma yazalım. Serinin tanımında ilk elemanın 0 ve ikinci elemanın 1 olduğu belirtilmişti. Bu doğrultuda, programımızda üç farklı değişken (ilk, iki, sonraki) tanımlayıp programın başlangıcında ilk değişkenine 0 ve iki değişkenine 1 değeri atıyoruz. Sonra, ilk ve son elemanlarını toplayıp serinin tanımında verildiği gibi bir sonraki elemanın değerini hesaplıyoruz. Sonraki iterasyona geçmeden önce ilk ve iki değişkenlerimizin değerini güncelliyoruz. Böylece, bir döngü içerisinde istenilen sayı kadar Fibonacci serisindeki elemanları hesaplayabiliyoruz.

```
#include <stdio.h>

int main()
{
    int i, n;
    int ilk, iki, sonraki;

    ilk = 0;
    iki = 1;

    printf("İlk kaç tane elemanı istiyorsunuz : \n");
    scanf_s("%d", &n);

    printf("Fibonacci serisindeki ilk %d eleman : \n", n);

    printf("%d\n", ilk);
    printf("%d\n", iki);

    for (i = 2; i < n; i++)
    {
        sonraki = ilk + iki;

        ilk = iki;
        iki = sonraki;

        printf("%d\n", sonraki);
    }

    return 0;
}
```

Küçült-Fethet (Decrease&Conquer) Yöntemi ile Algoritma Tasarlama

Bu algoritma tasarım yönteminde, problemin tamamının çözümüyle küçük bir parçasının çözümü arasında bir bağlantı vardır. Problemin çözümü aşağıdan yukarıya ya da yukarıdan aşağıya olabilir. Tanıma bakacak olursak problemin özyinelemeli fonksiyon uygulaması ile çözüleceğini düşünsek de döngü-tekrar algoritmaları ile de çözüme ulaşabiliriz (Levitin, 2012). Bunun için verilebilecek örneklerden biri araya sokma sıralama algoritmasıdır. Bu sıralama algoritması iskambil kâğıtlarını sıraladığımız algoritmaya benzemektedir. Şekil 4.3'te bu algoritmanın çalışmasına bir örnek verilmiştir. Bu algoritmda, problemi birer birer küçülterek çözüme ulaşıyoruz. Örneğin de anlaşılacağı üzere, her iterasyonda bir tane elemanı, o anki sıralı dizide yerine koymaktayız. k 'inci iterasyonda dizinin ilk k tane elemanı kendi içerisinde sıralıdır. $k+1$ aşamada $A[k+1]$ elemanını, var olan sıralı dizimizde uygun yere koymak. Böylece dizideki eleman sayısı kadar iterasyon geçtiğinde, dizimiz sıralı hale gelir.

Sekil 4.3

Araya sokma sıralama algoritması örneği

13	34	10	25	44	22	15	90
13	34	10	25	44	22	15	90
10	13	34	25	44	22	15	90
10	13	25	34	44	22	15	90
10	13	25	34	44	22	15	90
10	13	22	25	34	44	15	90
10	13	15	22	25	34	44	90
10	13	15	22	25	34	44	90

Araya sokma sıralama algoritmasının C programlama dilindeki uygulaması aşağıda verilmiştir.

```
#include <stdio.h>

void main()
{
    int j, P, Tmp, N;
    int A[] = { 13, 34, 10, 25, 44, 22, 15, 90 };

    N = 8;

    for (P = 1; P < N; P++)
    {
        Tmp = A[P];

        for (j = P; j > 0 && A[j - 1] > Tmp; j--)
        {
            A[j] = A[j - 1];
        }

        A[j] = Tmp;
    }

    printf("Sıralamadan sonra dizi : ");
    for (P = 1; P < N; P++)
    {
        printf(" %d", A[P]);
    }
}
```

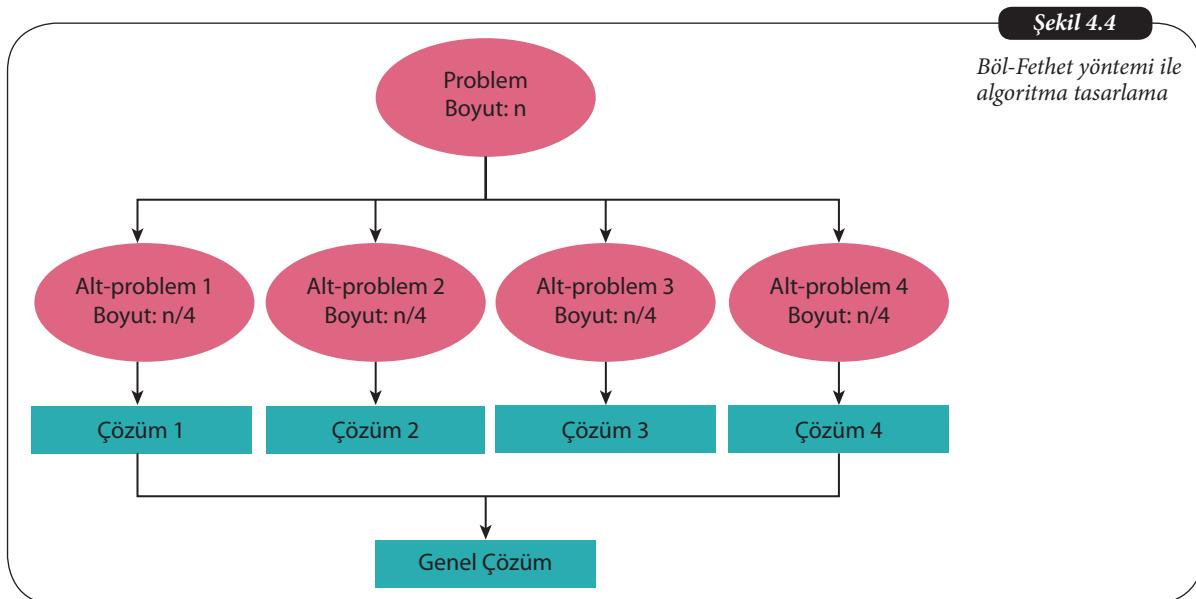
Küçült-Fethet algoritma tiplerinin farklı türleri vardır. Araya sokma sıralama algoritmasında problemin çözümüne birer birer yaklaşılıdı. Her adımda bir tane sayı, o anki sıralı yerine koymuldu. Farklı uygulamalar için problemin çözümüne, farklı veya değişken sabitlerle azaltma yaparak da ulaşılabilir.

Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet (Divide & Conquer) Yöntemi ile Algoritma Tasarımı

Böl-Fethet yöntemi en iyi bilinen algoritma tasarım yöntemlerinden biridir. Böl-Fethet yöntemindeki algoritmalar aşağıdaki gibi işleyiş yapısına sahiptir:

1. Öncelikli olarak problem genellikle eşit büyüklükteki alt parçalara ayrılır.
2. Her bir alt problem, genellikle özyinelemeli fonksiyon aracılığı ile çözülür.
3. Bütün alt problemlerin çözümü birleştirilerek genel sonuç elde edilir.

Örnek olarak problemimizin dört parçadan olduğunu varsayıyalım. Bu problemin Böl-Fethet yöntemi ile çözümü Şekil 4.4'te verilmiştir. Her bir alt problem teker teker çözülür. Alt problemlerin çözümünden sonra elimizdeki dört çözüm birleştirilir ve ana çözümü elde etmiş oluruz.



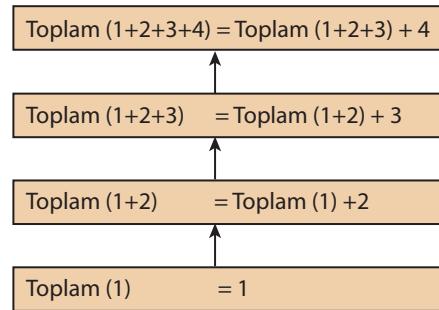
Böl-Fethet yönteminde öncelikli olarak 1'den N 'ye kadar olan sayıların toplamını hesaplayan bir program yazalım. Bu problemi özyinelemeli fonksiyon olarak yazalım:

$$Toplam(N) = \begin{cases} 1 & N=1 \\ Toplam(N-1)+N & N>1 \end{cases}$$

Sayısal örnek olarak 1'den 4'e kadar olan sayıların toplamını hesaplayan programı düşünelim. Bu problemi, 1'den 3'e kadar olan sayıların toplamına 4'ü ekleyerek bulabiliriz. Bunu özyinelemeli olarak düşündüğümüzde Şekil 4.5'teki yapı karşımıza çıkmaktadır. Problemin en küçük parçasına geldiğimizde, yani 1'e kadar olan sayıların toplamını hesapladıkten sonra geriye doğru gelerek bulduğumuz sonuçları birleştirerek ana sonuca ulaşmaktadır.

Sekil 4.5

1'den 4'e kadar sayıların toplamı



Bu programın C kodunu aşağıdaki gibi yazabiliriz:

```
#include <stdio.h>

int ToplamRecursive(int n)
{
    int tmpToplam = 0;

    if (n == 1) return 1;

    tmpToplam = ToplamRecursive(n - 1);

    return tmpToplam + n;
}

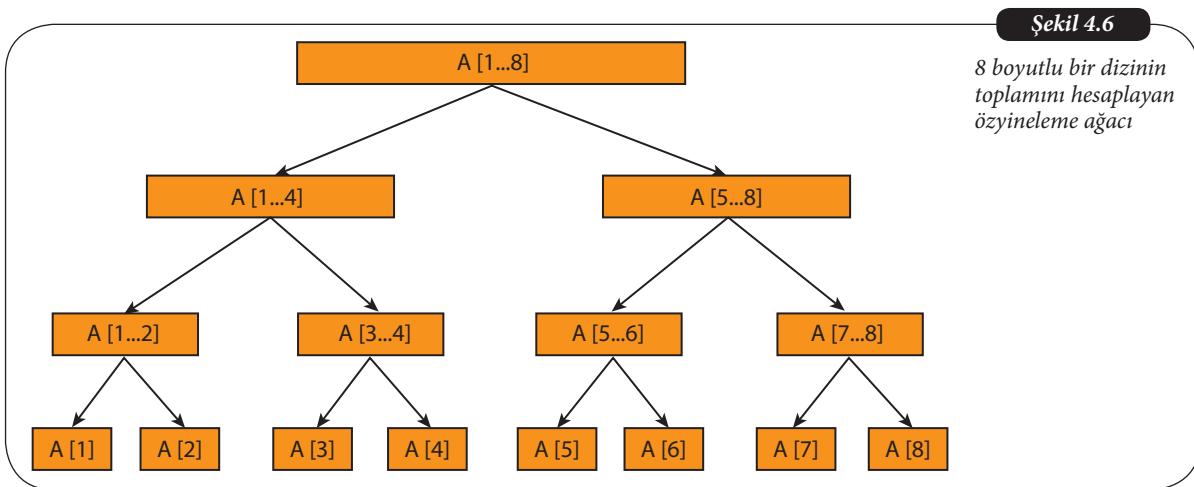
void main()
{
    int x = 0;

    x = ToplamRecursive(4);

    printf("Toplam: %d\n", x);

    return 0;
}
```

İkinci örneğimizde daha önce tekrarlamalı olarak çözümünü yaptığımız bir dizinin elemanlarının toplamını hesaplayan programı, bu defa özyinelemeli fonksiyon aracılığı ile gerçekleştireceğiz. Bu problemin çözümünde, diziyi iki bölüme ayırarak her bölümdeki elemanların toplamını ayrı ayrı hesaplayıp, bu iki toplamı birleştirerek ana sonuca ulaşabiliyoruz. Şekil 4.6'da dizideki eleman sayısı 8 olduğundaki özyineleme ağacının yapısı görülmektedir. Problemin çözümüne, ağacın yapraklarındaki sonuçlar birleştirilerek başlanır ve birleştirme işlemi ağacın kökünde son bulur. Böylece genel çözüm hesaplanmış olur.



Özyinelemeli yapısı Şekil 4.6'da verilen programın uygulaması ise aşağıda verilmiştir:

```
#include <stdio.h>

int DiziToplam(int A[], int N)
{
    if (N == 1) return A[0];

    int orta = N / 2;
    int localSum1 = DiziToplam(&A[0], orta);
    int localSum2 = DiziToplam(&A[orta], N - orta);

    return localSum1 + localSum2;
}

int main()
{
    int a;
    int A[] = { -1, 13, 17, -6, 0, 18, 20, 12 };

    a = DiziToplam(A, 8);

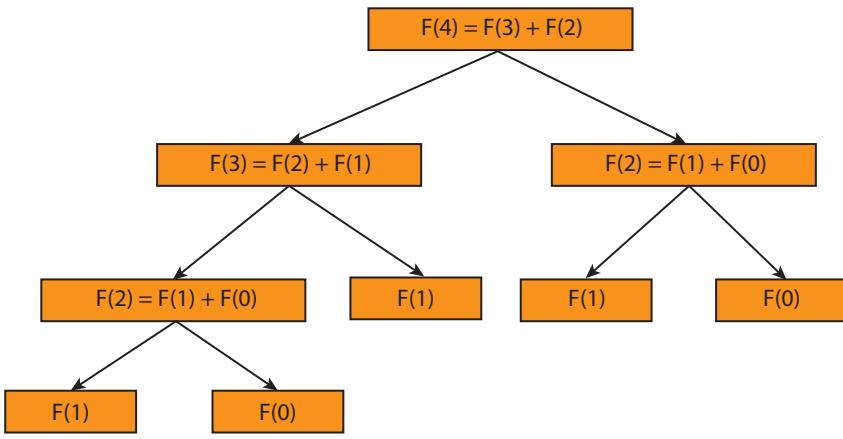
    printf("Dizinin elemanlarının toplamı : %d\n", a);
}
```

Özyinelemeli fonksiyonlar ve Böl-Fethet yöntemindeki son örneğimiz Fibonacci dizisinin elemanlarını hesaplayan fonksiyondur. Daha önce ifade edildiği üzere Fibonacci dizisinde bir eleman kendinden önce gelen iki elemanın toplamıdır. Bu durumu, özyinelemeli fonksiyon olarak şu şekilde ifade edebiliriz:

$$F(n) = \begin{cases} n & n \leq 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Şekil 4.7

Dördüncü Fibonacci sayısını hesaplayan özyinelemeli fonksiyon ağacı



Dördüncü Fibonacci sayısının özyinelemeli fonksiyon ağacı Şekil 4.7'de verilmiştir. Bu ağaçtan da görüldüğü üzere ağacımızı oluştururken taban duruma gelince ağacımızın yaprakları oluşmuş olur. Bu dizi için taban durumumuz, dizinin 0. ve 1. elemanlarıdır. Bundan sonra gelecek bütün elemanları, bu elemanlara bağlı olarak hesaplayabiliriz. Fibonacci dizisinin C kodu aşağıda verilmiştir:

```

#include <stdio.h>

int fibonacci(int z)
{
    if (z <= 1)
    {
        return z;
    }
    else
    {

        return fibonacci(z - 1) + fibonacci(z - 2);
    }
}

void main()
{
    int i, n;

    printf("İlk kaç tane elemanı istiyorsunuz : \n");
    scanf_s("%d", &n);

    for (i = 0; i < n; i++)
    {
        printf("%d , fibonacci(i));
    }
}
  
```

SIRA SİZDE



Verilen bir ikili ağactaki eleman sayısını hesaplayan özyinelemeli fonksiyonu yazınız.

Özet



Algoritmaların hesaplamadaki rolünü açıklamak

Kitabımızın bu ünitesinde öncelikle algoritmaların hesaplamadaki ve problem çözmedeki rolü anlatıldı. Matematiksel olarak çözüleceğimiz her problemi algoritmalar ile çözebiliriz.



Algoritma ile problem çözmenin genel aşamalarını listelemek

Matematiksel olarak ifade edebileceğimiz problemleri algoritmalarla çözebiliriz. Algoritma tasarım ve analizinin genel yapısı birkaç aşamadan oluşmaktadır. Bunlar problemi anlamak, algoritma tasarım teknüğine karar vermek, algoritmayı tasarlamak, doğruluğunu kanıtlamak, algoritmayı analiz etmek ve uygulamayı geliştirmektir. Ünitemizde çeşitli algoritma tasarım teknikleri incelendi ve algoritmalarımızın uygulama örnekleri verildi.



Tekrarlamalı döngülerle algoritma tasarımını özetlemek

Algoritma analiz yöntemlerinden öncelikle döngü ve tekrarlar ile algoritma tasarım yöntemi incelendi. Döngü algoritmaları ile bir dizideki en büyük eleman nasıl bulunacağı, Fibonacci dizisinin elemanlarının nasıl hesaplanacağı açıklandı.



Küçült-Fethet yöntemi ile algoritma tasarımını açıklamak
İkinci tasarım yöntemi olarak Küçült-Fethet tekniği anlatıldı. Bu teknikte problemin çözümü yukarıdan yukarıya ya da yukarıdan aşağıya doğru olabileceği ifade edildi. Bu teknikte, problemin küçük parçalarını adım adım çözerek sonuca ulaşmaktayız. Bu teknigue örnek olarak, araya koyma sıralama algoritması verilebilir.



Böl-Fethet yöntemi ile algoritma tasarımını açıklamak
Son teknik olarak Böl-Fethet yöntemi ele alındı. Bu yöntemde, problem genellikle eşit boyutlardaki parçalara bölünmekte ve her bir parçanın çözümü bulunduktan sonra bulunan sonuçlar birleştirilerek ana çözüme ulaşılmaktadır. Bu teknikte genellikle özyinelemeli fonksiyonlar kullanılır. Bu fonksiyonlarda problemin en küçük parçası için çözüm üretilir ve bu çözümden genel sonuca varılır. Algoritmaları tasarladıkten sonra, tasarlampış olduğumuz algoritmanın doğruluğunu teyit etmemiz, algoritmayı çalışma zamanı ve hafıza gereksinimi açısından analiz etmemiz gerekmektedir.

Kendimizi Sınayalım

- 1.** Fibonacci dizisinin ilk elemanı 0'dan başladığında bu dizinin beşinci elemanı ne olur?
 - 1
 - 3
 - 13
 - 21
 - 34

- 2.** Fibonacci dizisinin ilk elemanı 0'dan başladığında, bu dizinin 4. elemanı için özyinelemeli fonksiyon ağacını çizdğimizde, $F(4)$ değeri aşağıdakilerden hangisine eşit olur?
 - $F(2) + F(3)$
 - $F(1) + F(2)$
 - $F(0) + F(1)$
 - $F(1) + F(4)$
 - $F(3) + F(4)$

- 3.** 1'den N'ye kadar olan sayıların toplamı için yazılan özyinelemeli fonksiyonlardan hangisi doğrudur? ($\text{Toplam}(N) = 1 + 2 + 3 + \dots + N$)
 - $\text{Toplam}(N) = \text{Toplam}(N-1) + N$
 - $\text{Toplam}(N) = \text{Toplam}(N-2) + N$
 - $\text{Toplam}(N) = \text{Toplam}(N-1) + 1$
 - $\text{Toplam}(N) = \text{Toplam}(N-2) + 2N$
 - $\text{Toplam}(N) = \text{Toplam}(N-1) + 2N$

- 4.** Fibonacci serisi için $n > 1$ değerlerinde özyinelemeli fonksiyon aşağıdakilerden hangisidir?
 - $F(n) = F(n-1) + F(n-3)$
 - $F(n) = F(n-1) + F(n-1)$
 - $F(n) = F(n-1) + 2F(n-2)$
 - $F(n) = 3F(n-1) + F(n-2)$
 - $F(n) = F(n-1) + F(n-2)$

- 5.** Algoritma tasarım ve analiz aşamalarının ilki aşağıdakilerden hangisidir?
 - Problemi anlamak
 - Tekniğe karar vermek
 - Uygulamayı gerçekleştirmek
 - Doğruluğunu kanıtlamak
 - Algoritma analizi yapmak

- 6.** Algoritma tasarımları ve analizi sürecinde, algoritmayı tasarladıkten sonraki aşama aşağıdakilerden hangisidir?
 - Problemi anlama
 - Tekniğe karar verme
 - Uygulamayı gerçekleştirmek
 - Doğruluğunu kanıtlama
 - Algoritma analizi yapma

- 7.** Araya sokma sıralama algoritmasında başlangıçtaki dizi ve birinci adım sonucu aşağıda verilmiştir. Buna göre ikinci adımda elemanların sıralaması nasıl olur?

Başlangıç: 13 | 34 10 25 44 22 15 90
Adım : 13 34 | 10 25 44 22 15 90

 - 10, 13, 34, 25, 44, 22, 15, 90
 - 13, 10, 34, 25, 44, 22, 15, 90
 - 13, 34, 10, 25, 44, 22, 15, 90
 - 10, 13, 34, 25, 44, 22, 90, 15
 - 34, 13, 10, 25, 44, 22, 15, 90

- 8.** Böl-Fethet yöntemi aşamalarının doğru sıralaması aşağıdakilerden hangisidir?
 - Problemi eşit parçalara ayırmak
 - Genel sonucu kullanıcıya sunmak
 - Alt parçaların sonucunu birleştirmek
 - Alt parçaların çözümünü yapmak
 - I, II, III ve IV
 - I, III, II ve IV
 - I, IV, III ve II
 - III, II, I ve IV
 - III, II, IV ve I

- 9.** Aşağıda verilen "Gizem" fonksiyonu hangi değeri hesaplamaktadır?

```
int Gizem(int n)
{
    int tmp = 0;
    if (n == 1) return 1;
    tmp = Gizem(n - 1);
    return tmp + n;
}
```

- 1'den N'ye kadar olan tek sayıların toplamı
- 1'den N'ye kadar olan sayıların toplamı
- 1'den N'ye kadar olan çift sayıların toplamı
- Fibonacci dizisinin elemanları
- Bir dizideki elemanların toplamı

Kendimizi Sınayalım Yanıt Anahtarı

- 10.** Aşağıda verilen "Sir" fonksiyonu hangi değeri hesaplamaktadır?

```
int Sir(int A[], int N)
{
    int i;
    int tmp=0;

    for (i = 0; i < N; i++)
    {
        if (A[i]>0)
            tmp = tmp + A[i];
    }

    return tmp;
}
```

- a. Dizideki tüm sayıların toplamını
- b. Dizideki negatif sayıların toplamını
- c. Dizideki pozitif sayıların toplamını
- d. Dizideki eleman sayısını
- e. Dizideki en büyük elemanı

- | | |
|-------|---|
| 1. b | Yanıtınız yanlış ise "Döngü-Tekrarlama Algoritmaları" konusunu yeniden gözden geçiriniz. |
| 2. a | Yanıtınız yanlış ise "Küçült-Fethet Yöntemi ile Algoritma Tasarlama" konusunu yeniden gözden geçiriniz. |
| 3. a | Yanıtınız yanlış ise "Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet Yöntemi ile Algoritma Tasarımı" konusunu yeniden gözden geçiriniz. |
| 4. e | Yanıtınız yanlış ise "Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet Yöntemi ile Algoritma Tasarımı" konusunu yeniden gözden geçiriniz. |
| 5. a | Yanıtınız yanlış ise "Algoritma ile Problem Çözme" konusunu yeniden gözden geçiriniz. |
| 6. d | Yanıtınız yanlış ise "Algoritma ile Problem Çözme" konusunu yeniden gözden geçiriniz. |
| 7. a | Yanıtınız yanlış ise "Küçült-Fethet Yöntemi ile Algoritma Tasarlama" konusunu yeniden gözden geçiriniz. |
| 8. c | Yanıtınız yanlış ise "Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet Yöntemi ile Algoritma Tasarımı" konusunu yeniden gözden geçiriniz. |
| 9. b | Yanıtınız yanlış ise "Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet Yöntemi ile Algoritma Tasarımı" konusunu yeniden gözden geçiriniz. |
| 10. c | Yanıtınız yanlış ise "Döngü-Tekrarlama Algoritmaları" konusunu yeniden gözden geçiriniz. |

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

Diziye yeni bir sayı eklemek istediğimizde karşılaşabileceğimiz özel durum, dizinin kapasitesinin dolu olmasıdır. Örneğin, dizinin kapasitesi 100 iken dizİYE 101. elemanı eklemek istememiz özel bir durum oluşturacaktır.

Sıra Sizde 2

$ax^2 + bx + c$ şeklinde verilen ikinci dereceden bir denklemin köklerini bulmak için aşağıdaki algoritma adımlarını izleyebiliriz:

1. a, b, c değerlerini kullanıcıdan al
2. delta=b*b-4ac
3. delta < 0 ise reel kök yoktur yaz ve 6. adıma git
4. x1=b-kök(delta)/2a, x2=-b-kök(delta)/2a
5. x1 ve x2 değerlerini ekrana yaz
6. Dur

Yararlanılan ve Başvurulabilecek Kaynaklar

Sıra Sizde 3

```
int minimum(int A[], int N)
{
    int i;
    int min;

    min = A[0];

    for (i = 1; i < N; i++) {
        if (min > A[i])
            min = A[i];
    }

    return min;
}
```

Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) **Introduction to Algorithms**, 3rd Edition, MIT Press.
Levitin A. (2012) **Introduction to The Design & Analysis of Algorithms**, 3rd Edition, Pearson.
Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

Sıra Sizde 4

```
int count(node* kok)
{
    if (kok == NULL)

        return 0;
    else if (kok->sol == NULL && kok-
>sag == NULL)

        return 1;
    else

        return count(kok->sol) +
count(kok->sag) + 1;
}
```


5

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Algoritma analizinin önemini açıklayabilecek,
 - 🕒 Verilen bir algoritmanın analizini yapabilecek,
 - 🕒 Algoritmaların en kötü, en iyi ve ortalama durumdaki verimliliklerinin farkını özetleyebilecek,
 - 🕒 Asimptotik gösterimleri uygulayabilecek,
 - 🕒 Özyinelemeli olan ya da olmayan algoritmaların analizini yapabilecek bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- Algoritma Analizi
- Asimptotik Gösterimler
- Özyinelemeli Fonksiyon
- Çalışma Zamanı
- Zaman Karmaşıklığı

İçindekiler

Algoritmalar ve Programlama

Algoritma Analizi

- Giriş
- ALGORİTMA ANALİZİ İÇİN MATEMATİKSEL ARKA PLAN
- FONKSİYONLARIN BüYÜMESİ
- ALGORİTMALARIN EN KÖTÜ DURUM, EN İYİ DURUM VE ORTALAMA DURUM VERİMLİLİKLERİ
- ASİMPTOTİK GÖSTERİMLER
- ÖZYİNELEMELİ OLMAYAN ALGORİTMALARIN ANALİZİ
- ÖZYİNELEMELİ ALGORİTMALARIN ANALİZİ

Algoritma Analizi

GİRİŞ

Kitabımızın önceki ünitesinde farklı algoritma tasarım yöntemleri anlatılmıştı. Bu üniteımızde ise algoritma analizi konusu ele alınacaktır. Bir problem için algoritma tasarlarken bu programı bilgisayarda çalıştırduğumızda, problemin çözümüne ulaşmasının ne kadar zaman alacağı önemli bir husustur. Bilindiği üzere, bir problemin farklı algoritmalarla birden fazla çözümü olabilir. Bir problemi çözmek istediğimizde, olası algoritmalarдан çalışma zamanı ve bellek gereksinimi açısından uygun olanını seçmek oldukça önemlidir. Örneğin, sıralama problemi için birçok algoritma kullanılabilir. Sıralama algoritmasını küçük bir veri seti üzerinde çalışıracaksak, programcının en kolay tasarlayacağı algoritmayı seçmesi en basit yaklaşımındır. Ancak sıralama probleminde milyonlarca veri olması söz konusuya çözüme makul bir zaman içinde ulaşılması tercih edilecektir. Böyle durumlarda, çalışma zamanı en kısa olan algoritmayı seçmemiz kendi yararımıza olacaktır. Ayrıca, kullanılacak algoritmanın bellek gereksinimi de üzerinde dikkatle düşünülmesi gereken konulardan biridir.

Algoritmaları çalışma zamanı açısından karşılaştırmak istediğimizde aklımıza gelebilecek temel ölçütlerden biri, ilgili algoritmanın çözümüne kaç saniyede ulaşıldığıdır. Ancak, saniye, dakika, saat vb. zaman birimlerinin bu ölçümde ölçüt olarak kullanılması çok uygun değildir. Herhangi bir problemin kaç saniyede çözüme ulaşabileceğini, kullandığımız bilgisayarın konfigürasyonuna ve kullandığımız derleyicinin ürettiği makine koduna bağlı olarak değiştirmektedir. Çalışma zamanı karşılaştırması için kullanılabilecek diğer bir yaklaşım ise algoritmadaki bütün operasyonları tek tek saymaktır. Ancak bu işlem hem zor hem de gereksizdir. Bunun yerine, algoritmanın içerisinde temel işlemin yapıldığı yeri bulup buradaki operasyon sayısını hesaplamak çok daha uygun bir yaklaşım olacaktır. Dikkatle incelendiği takdirde, bir algoritmanın temel operasyonunu bulmak çok zor değildir. Örneğin bir sıralama algoritmasında temel operasyon, sayıların karşılaştırıldığı kısımdır. Sıralama algoritmasının ne kadar karşılaştırma operasyonu yaparak ana sonuca ulaştığı, temel operasyonu oluşturmaktadır.

Bu ünitemizde, algoritmada temel operasyonun analizini yapacağız. Ünitemizde kullanılacak temel kavramları şu şekilde açıklayabiliriz:

Çalışma Zamanı (running time): Tasarlanan algoritma ile problemin çözümüne ulaşabilmek yapılan toplam temel operasyon sayısıdır. $T(n)$ ile ifade edilir. Temel operasyonlar, karşılaştırma sayısı, döngü içerisinde dönme sayısı vb. işlemler olabilir.

Zaman Karmaşıklığı (time complexity): Bir algoritmanın verilen asimptotik gösterime göre karmaşıklık derecesini gösterir.

Alan Karmaşıklığı: Algoritmanın eleman sayısının çok büyük olduğu durumlarda, problemin çözümünü ulaşabilmeye yönelik bellek gereksinimidir.

ALGORİTMA ANALİZİ İÇİN MATEMATİKSEL ARKA PLAN

Algoritma analizi ile uğraşırken, bazı hesaplamaları yapabilmek için çeşitli matematiksel işlemlere hâkim olunması gerekmektedir. Bu sebeple, algoritma analizinde ve hesaplamalarda kullanılacak olan bazı temel ifadeler ve formüller aşağıda verilmiştir:

- 1'den N'ye kadar olan sayıların toplamı

$$S(N) = 1 + 2 + 3 + 4 + \dots + N = \sum_{i=1}^N i = \frac{N(N+1)}{2}$$

- 1'den N'ye kadar olan sayıların karelerinin toplamı

$$\sum_{i=1}^N i^2 = \frac{N * (N+1) * (2N+1)}{6} \approx \frac{N^3}{3}$$

- Geometrik seri toplamı

$$\sum_{i=0}^N A^i = \frac{A^{N+1} - 1}{A - 1}, A > 1$$

$$\sum_{i=0}^N A^i = \frac{1 - A^{N+1}}{1 - A} = \Theta(1), A < 1$$

- Harmonik seri toplamı

$$H_n = \sum_{i=1}^n \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = (\ln n) + O(1)$$

- Logaritmalar

$$\log A^B = B * \log A$$

$$\log(A * B) = \log A + \log B$$

$$\log\left(\frac{A}{B}\right) = \log A - \log B$$

- a'dan b'ye kadar olan değerler için fonksiyonun değerler toplamı

$$\sum_{i=a}^b f(i) = \sum_{i=0}^b f(i) - \sum_{i=0}^{a-1} f(i)$$

- Toplamların parçalı hesaplanması

$$\sum_{i=1}^n (4i^2 - 6i) = 4 \sum_{i=1}^n i^2 - 6 \sum_{i=1}^n i$$



FONKSİYONLARIN BüYÜMESİ

Tasarladığımız algoritmaları analiz ederken, az sayıda girdi için sonucun ne kadar hızlı hesapladığını incelemek çok anlamlı değildir. Örneğin, yalnızca 10 tane sayıyı sıralayacak sıralama algoritmalarını düşünelim. Böyle bir işlem için gerek kabarcık sıralaması gerek birleştirme sıralaması algoritması çok kısa bir sürede sonuç verecektir. Başka bir ifadeyle, girdi sayısı az olduğunda iki algoritma arasındaki farkı net bir şekilde göremeyiz. Ancak, 10 tane yerine 1 milyon tane sayıyı sıralamak istediğimizi düşünürsek, böyle bir durumda girdi miktarı oldukça büyük olduğu için iki algoritmanın zaman karmaşıklığı arasındaki fark kolayca anlaşılacaktır. Böylelikle, zaman karmaşıklığı daha düşük olan birleştirme sıralama algoritması çok daha çabuk sonuca ulaşacaktır.

İkinci bir örnek verecek olursak, 1000. Fibonacci sayısını hesaplamak, beşinci Fibonacci sayısını hesaplamaktan daha uzun zaman alacaktır. Bir algoritmanın zaman karmaşıklığı, girdi sayısının yeterince büyük olduğu varsayılarak hesaplanmaktadır. Tablo 5.1'de algoritma analizinde sıkılıkla kullanılan fonksiyonların girdi değerlerine karşılık gelen çıktıları verilmiştir. Fonksiyona bağlı olarak, girdi değeri büyüdükle sonucun ne kadar farklılığı bu tablodan rahatlıkla anlaşılabilir. Kabarcık sıralaması n^2 zaman alırken, birleştirme sıralaması $n \log n$ zaman almaktadır. Bu durumda, 1 milyon tane sayıyı sıralamak istediğimizde, kabarcık sıralaması 10^{12} birim zaman alacakken, birleştirme sıralaması 2×10^7 birim zaman olacaktır. Çalışma zamanındaki farklılık, bu verilerden kolayca anlaşılabilir. Tablodan da görüldüğü gibi, üstel ve faktöriyel fonksiyonlar çok hızlı büyümektedir. Üstel ya da faktöriyel karmaşıklığa sahip algoritmalar çok yavaş çalışacaktır. Bu nedenle, üstel karmaşıklığa sahip algoritmalar yalnızca az girdi sayısına sahip problemler için kullanışlıdır.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	3.3x10	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10^4	10^6	10^9		
10^4	13	10^4	1.3×10^5	10^8	10^{12}		
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}		
10^6	20	10^6	2×10^7	10^{12}	10^{18}		

Tablo 5.1
Algoritma Analizindeki
Önemli Fonksiyonların
Bazı Yaklaşık Değerleri
(Levitin, 2012)

Bir algoritmanın girdisi $n=2^{10}$ olduğunda Tablo 5.1'de verilen fonksiyonların çıktı değerlerini hesaplayınız.



SIRA SİZDE

ALGORİTMALARIN EN KÖTÜ DURUM, EN İYİ DURUM VE ORTALAMA DURUM VERİMLİLİKLERİ

Algoritmaların durum verimlilikleri verilen girdiye göre değişimlemektedir. Örneğin, bir lineer sıralama algoritmasını ele alalım. Bu algoritma, verilen bir sayının dizideki elemanlardan biri olup olmadığını sonucunu döndürmektedir. Bu algoritmda verilen sayı, dizideki ilk elemandan başlayarak sırasıyla dizinin tüm elemanlarıyla karşılaştırılmaktadır. Bulmak istediğimiz değerin dizideki ilk elemana eşit olması, arama probleminde karşılaşabileceğimiz en iyi durumdur. Böyle bir durumda, dizinin ilk elemanı ile karşılaştırmaya hemen sonuca ulaşıp dizinin diğer elemanlarıyla karşılaştırmaya gerek kalmayacaktır. Bu durum, lineer arama algoritmasının optimal girdi için verimlidir. Lineer arama algoritması aşağıda verilmiştir:

```

int lineer_arama(int A[], int n, int arama)
{
    int c;
    int sonuc = -1;

    for (c = 0; c < n; c++)
    {

        if (A[c] == arama)
        {

            sonuc = c;
            break;
        }
    }
    return sonuc;
}

```

Lineer arama örneğinden de anlaşılacağı üzere bir algoritmanın en iyi durum verimliliği, en optimal girdi durumundaki algoritma karmaşıklığıdır. Ancak en iyi durum verimliliği çok fazla anlam ifade etmemektedir. Çünkü çoğu zaman verilen girdiler sürekli değişmektedir ve optimal girdinin denk gelme olasılığı oldukça düşüktür. Karşılaşabileceğimiz olası bir başka durum ise girdi verisinin algoritma için en kötü olduğu durumdur. Lineer arama algoritmasını düşünerek olursak aradığımız elemanın dizinin son elemanı olması veya dizide hiç bulunmaması karşılaşabileceğimiz en kötü girdi durumlarıdır. Bu durumlarda, lineer arama algoritması ilk elemandan son elemana kadar sırasıyla bütün elemanlar için arama işlemini gerçekleştirecektir. Başka bir ifadeyle, en kötü durum verimliliği söz konusudur. Algoritmaların karmaşıklığını hesaplarken genellikle en kötü durumu göz önüne almaktayız.

Son olarak bakacağımız durum ise ortalama durum verimliliğidir. Burada ise algoritmamızı farklı girdiler için pek çok kez çalıştırarak çalışma sürelerinin ortalamasını alırız. Böylece, ortalama durum verimliliğini elde etmiş oluruz. Lineer arama algoritmasını düşünerek olursak verilen girdiye göre 1., 2., 3., ..., N. gibi adımlarda aradığımız sayıyı bulabiliriz. Bu olasılıkların hepsinin ortalamasını alduğumuzda ($N/2$)'nci adımda, yani dizinin orta elemanında sonucu bulmamız gerekmektedir. Bu değer ise lineer arama algoritmasının ortalama durum verimliliğidir.

SIRA SİZDE



3

Bir dizinin en büyük elemanını bulma algoritmasındaki en iyi durum ve en kötü durum verimliliği nedir?

ASİMPTOTİK GÖSTERİMLER

Bir algoritma çalışlığında kaç birim adımda sonuca ulaşacağı hesaplanarak aynı problemi çözen farklı algoritmaların verimliliği karşılaştırılabilir. Bilgisayar bilimcileri, fonksiyonların büyümelerini de göz önünde bulundurarak algoritmaları karşılaştırırken kullanılmak üzere üç tane gösterim tanımlamıştır:

- Büyük O Gösterimi
- Büyük Ω Gösterimi
- Büyük Θ Gösterimi

Bu gösterimlere göre, algoritmaları kendi arasında verimlilik açısından karşılaştırabiliriz.

Büyük O Gösterimi

Matematiksel gösterimin dışında tanımlayacak olursak büyük O gösterimini kullandığımızda analiz ettiğimiz algoritmanın çalışma zamanının belirli bir girdi değerinden sonra bu gösterimdeki fonksiyondan daha küçük olarak çalıştığını garanti etmektedir. Başka bir ifadeyle, algoritmamız $O(n^2)$ mertebesinde ise algoritmamız içerisindeki işlemlerin n^2 adımlına eşit ya da daha az adım gerektirdiğini göstermektedir. Büyük O gösterimiyle bir fonksiyonun üst sınırını belirtmiş oluruz. Büyük O gösterimi görsel olarak Şekil 5.1'de açıklanmaktadır. n_0 ve c pozitif katsayılar olmak üzere, O gösterimini şu şekilde tanımlayabiliriz:

$$O(g(n)) = \{f(n) : 0 \leq f(n) \leq cg(n), n \geq n_0\}$$

$f(n) = n + 6$ fonksiyonu $O(n)$ midir?

ÖRNEK 5.1

$f(n)$ fonksiyonunun $O(n)$ olduğunu gösterebilmek için aşağıdaki eşitsizliği sağlayacak n_0 ve c pozitif katsayılarını bulmamız gerekmektedir.

$$O(g(n)) = \{f(n) : 0 \leq f(n) \leq cg(n), n \geq n_0\}$$

Burada $n_0 = 2$ ve $c = 7$ alındığında, aşağıdaki eşitsizlik sağlanmaktadır.

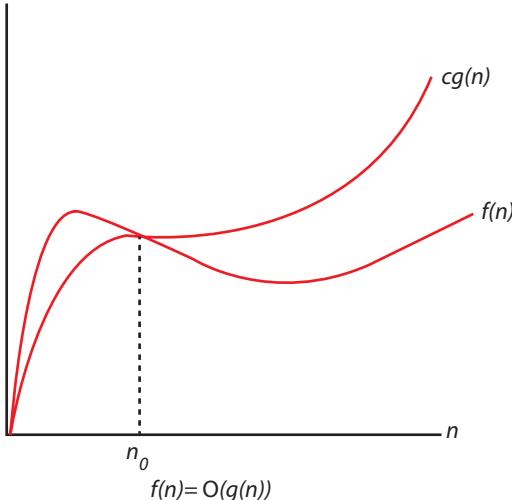
$$0 \leq f(n) \leq cg(n), n \geq n_0$$

$$0 \leq n + 6 \leq 7n, n \geq 2$$

Gördüğü üzere $f(n) = n + 6$ fonksiyonu $O(n)$ 'dır.

Şekil 5.1

Büyük O gösterimi



Büyük Ω Gösterimi

Algoritma analizinde kullanılan ikinci gösterim, büyük Ω gösterimidir. Bu gösterim ile verilen bir algoritmanın değerinin, Ω gösterimindeki fonksiyonun değerinden daha büyük olduğunu göstermektedir. Yani Ω gösterimi fonksiyonun alt sınırını belirtmektedir. Şekil 5.2'de Ω gösterim görsel olarak açıklanmaktadır. n_0 ve c pozitif katsayılar olmak üzere, Ω gösterimini şu şekilde tanımlayabiliriz:

$$\Omega(g(n)) = \{f(n) : 0 \leq cg(n) \leq f(n), n \geq n_0\}$$

ÖRNEK 5.2

$f(n) = 5n^2 - 3n$ fonksiyonu $\Omega(n^2)$ midir?

$f(n)$ fonksiyonunun $\Omega(n^2)$ olduğunu gösterebilmek için aşağıdaki eşitsizliği sağlayacak n_0 ve c pozitif katsayılarını bulmamız gerekmektedir.

$$\Omega(g(n)) = \{f(n): 0 \leq cg(n) \leq f(n), n \geq n_0\}$$

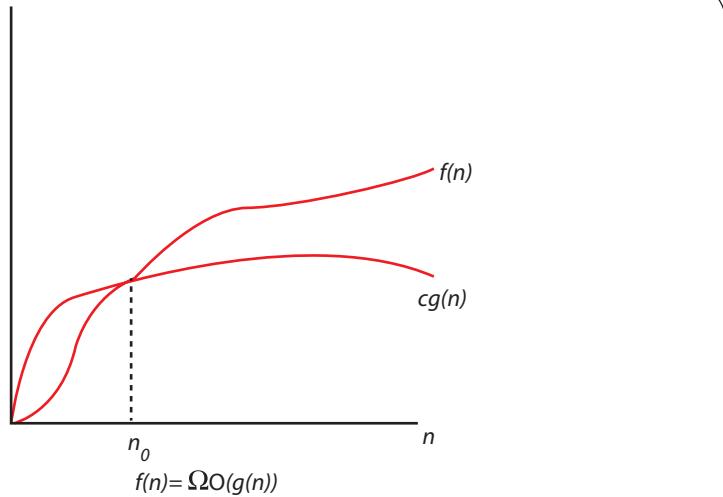
Burada $n_0 = 4$ ve $c = 4$ alındığında aşağıdaki eşitsizlik sağlanmaktadır.

$$\begin{aligned} 0 &\leq cg(n) \leq f(n) \quad n \geq n_0 \\ 0 &\leq 4n^2 \leq 5n^2 - 3n, \quad n \geq 4 \end{aligned}$$

Görüldüğü üzere $f(n) = 5n^2 - 3n$ fonksiyonu $\Omega(n^2)$ 'dir.

Şekil 5.2

Büyük Ω Gösterimi

**Büyük Θ Gösterimi**

Büyük Θ gösterim, daha önce incelenmiş olan O ve Ω gösterimlerinin birleşimine benzemektedir. Büyük Θ gösteriminde, algoritmanın fonksiyonunu alttan ve üstten sınırlayacak bir fonksiyon bulmak istenir. Büyük Θ gösterimi görsel olarak Şekil 5.3'te yer almaktadır. n_0 , c_1 ve c_2 pozitif katsayılar olmak üzere, Ω gösterimini şu şekilde tanımlayabiliriz:

$$\Theta(g(n)) = \{f(n): c_1 g(n) \leq f(n) \leq c_2 g(n), n \geq n_0\}$$

ÖRNEK 5.3

$f(n) = 2n + 5$ fonksiyonu $\Theta(n)$ midir?

$f(n)$ fonksiyonunun $\Theta(n)$ olduğunu gösterebilmek için aşağıdaki eşitsizliği sağlayacak n_0 , c_1 ve c_2 pozitif katsayılarını bulmamız gerekmektedir.

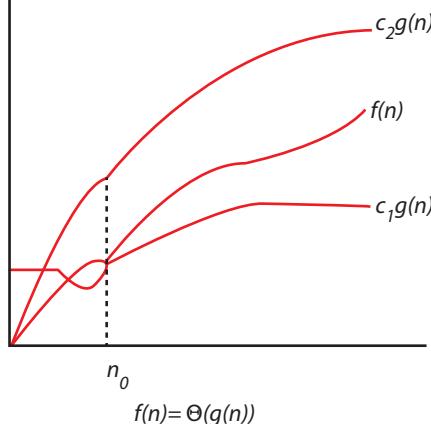
$$\Theta(g(n)) = \{f(n): c_1 g(n) \leq f(n) \leq c_2 g(n), n \geq n_0\}$$

Burada n_0 , $c_1 = 2$ ve $c_2 = 3$ alındığında aşağıdaki eşitsizlik sağlanmaktadır:

$$\begin{aligned} c_1 g(n) &\leq f(n) \leq c_2 g(n), \quad n \geq n_0 \\ 2n &\leq 2n + 5 \leq 3n, \quad n \geq 5 \end{aligned}$$

Görüldüğü üzere, $f(n) = 2n + 5$ fonksiyonu $\Theta(n)$ 'dir.

Şekil 5.3

Büyük Θ Gösterimi

Temel asimptotik verimlilik sınıfları Tablo 5.2'de verilmiştir. Sabit sınıfı genellikle en iyi durum verimliliklerinde karşımıza çıkmaktadır. Logaritmik sınıf algoritmalarında ise problemin büyüklüğü her bir iterasyonda belirli bir sayıya (genellikle 2) bölünmektedir. Lineer sınıfındaki algoritma, dizide verilen bütün elemanların üzerinden tek tek gitmektedir. Yarı doğrusal sınıfındaki algoritmalar çoğunlukla böl-birleştir algoritmalarıdır. İkinci derece (kareli) sınıfındaki algoritmalarda iki tane iç içe döngü bulunmaktadır. Üçüncü derece (kübik) sınıfındaki algoritmalarla üç tane iç içe döngü bulunmaktadır. Üstel sınıfındaki algoritmalar, bir kümenin bütün alt kümeleri üzerinde işlem yapmaktadır. Faktöriyel algoritmalarında bir kümenin bütün permutasyonları üzerinde işlem yapılmaktadır.

Sınıf	Adı
1	Sabit
$\log n$	Logaritmik
n	Lineer
$n \log n$	Yarı doğrusal
n^2	İkinci dereceden (kareli)
n^3	Üçüncü dereceden (kübik)
2^n	Üstel
$n!$	Faktöriyel

Tablo 5.2
Temel Asimptotik
Verimlilik Sınıfları

$f(n) = 2n + n$ fonksiyonu $\Theta(n^2)$ midir?



SIRA SİZDE

ÖZYİNELEMELİ OLMAYAN ALGORİTMALARIN ANALİZİ

Bu bölümde, algoritmaların matematiksel olarak analizinin nasıl yapılacağını inceleyeceğiz. Bir algoritmayı analiz etmek, genel olarak aşağıda verilen adımlardan oluşmaktadır (Levitin, 2012).

1. Problemin girdi büyüklüğünü veren parametre belirlenir.
2. Algoritmanın temel operasyonu belirlenir.
3. Temel operasyonun sadece girdi büyüklüğüne bağlı olarak mı değiştiği kontrol edilir. Eğer başka parametrelere göre de değişiyorsa bunlar belirlenir.

4. Temel operasyon için toplam ifadesi bulunur.
5. Toplam ifadeleri için verilen standart formüller ve kurallar kullanılarak algoritmanın ait olduğu verimlilik sınıfı bulunur.

Algoritma Analizi ile İlgili Genel Kurallar

Algoritmaların analizini yaparken aşağıda verilen dört durumun analizinden faydalanabiliriz.

for Döngüsü: *for* döngüsünün çalışma zamanı, içerisindeki operasyonların çalışma zamanının iterasyon sayısı ile çarpımı kadardır.

```
for ( i = 0; i < n; i++)
{
    m = m + i;
}
```

Yukarıda verilen döngünün içerisinde sabit zamanlı bir işlem yapıldığından, döngü n defa bu sabit işlemi yapmaktadır. Bu nedenle, *for* döngüsü için zaman karmaşıklığı $O(n)$ 'dır.

İç içe for döngüsü: Öncelikli olarak içteki döngünün analizi yapılır. Daha sonra dıştaki döngünün analizi yapılır. İç içe döngünün çalışma zamanı iki döngünün çalışma sayılarının çarpımına eşittir.

```
for ( i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        m = m + i;
    }
}
```

Bu örnekte, içteki döngü ve dıştaki döngü her defa çalışmaktadır. Bu nedenle bu iç içe döngünün zaman karmaşıklığı $O(n^2)$ 'dır.

İki tane arka arkaya for döngüsü: Arka arkaya iki tane döngü söz konusu olduğunda, iki döngünün de çalışma sayıları bulunur ve bu sayılar birbiriyile toplanır.

```
for (i = 0; i < n; i++)
{
    m = m + i;
}

for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        m = m + i;
    }
}
```

Bu örnekte ilk döngünün zaman karmaşıklığı $O(n)$, iç içe olan ikinci döngünün zaman karmaşıklığı $O(n^2)$ 'dir. Bu döngülerin peş peşe çalıştığındaki çalışma zamanı iki çalışma zamanının toplamına eşit olaktır. Bu nedenle toplam zaman karmaşıklığı $O(n + n^2) = O(n^2)$ 'dir.

if/else deyimi: if ya da else kısmındaki çalışma zamanından hangisi daha büyükse o dikkate alınır.

```
if (a < 5)
{
    for (i = 0; i < n; i++)
    {
        m = m + i;
    }
}
else
{
    m = m + 1;
}
```

Bu örnekte if kısmının çalışma zamanı $O(n)$ iken else kısmının çalışma zamanı $O(1)$ 'dır. if/else deyiminin bir bütün olarak zaman karmaşıklığı $O(n)$ olur.

Aşağıda verilen kod parçasının çalışma zamanının üst sınırını (zaman karmaşıklığı) nedir?



SIRA SİZDE

5

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
        m = m + i;

    for (j = 0; j < n; j++)
        m = m + j;
}
```

Algoritma Analiz Örnekleri

Bu bölümde, özyinelemeli olmayan bazı algoritmaların analizi yapılacaktır. İlk örneğimiz, bir önceki bölümde tasarladığımız bir dizinin en büyüğünü (maksimum) bulma algoritmasıdır. Bu algoritmanın girdileri, dizinin kendisi ve dizideki eleman sayısıdır. Algoritmanın temel operasyonu *for* döngüsünün yer aldığı kısımdır. Bu kısmı analiz ettiğimizde, algoritmanın karmaşıklığını belirlemiş oluruz.

```
int maksimum(int A[], int N)
{
    int i; //-----> 1
    int max; //-----> 1

    max = A[0]; //-----> 1

    for (i = 1; i < N; i++) { //-----> N

        if (max < A[i]) //-----> N

            max = A[i];
    }

    return max; //-----> 1
}
```

Bu algoritmada temel operasyon dâhil bütün adımları toplarsak $T(N) = 2N + 4$ adım işlem vardır. Bunu büyük O gösterimine çevirirsek maksimumu bulma işlemi $O(N)$ üst sınırına sahiptir.

İkinci örnek olarak, bir dizideki elemanların toplamını bulma işleminin analizi yapılacaktır. Bu algoritma maksimumu bulmaya benzemektedir. Girdiler, dizi ve dizideki eleman sayısıdır. Dizideki temel operasyon *for* döngüsünün bulunduğu kısımdır.

```
int Toplama(int A[], int N)
{
    int i; //-----> 1
    int toplam = 0; //-----> 1

    for (i = 0; i < N; i++) { //-----> N
        toplam += A[i]; //-----> N
    }

    return toplam; //-----> 1
}
```

Bu algoritmda toplam adım sayısı $T(N) = 2N + 3$ kadardır. Yani bu algoritmanın zaman karmaşıklığı $O(N)$ 'dir.

Diğer örneğimiz lineer arama üzerindedir. Bu algoritma, daha önce açıkladığımız algoritmaların biraz farklılık göstermektedir. Temel operasyon *for* döngüsünün bulunduğu kısımdır. Ancak *for* döngüsünün içerisinde arama yaparken sonuca ulaşıldığında döngüden çıkışmaktadır. Yani döngü en kötü durumda, n defa dönmemektedir. En iyi durumda ise ilk iterasyonda sonuca ulaşmaktadır. Ancak algoritma analizi yaparken en kötü durum dikkate alınmaktadır. Böylece, algoritmanın bu degerden daha uzun süre çalışmayaceği garanti edilmektedir. En kötü durum dikkate alarak inceleme yaptığımızda, bu algoritmadaki toplam adım sayısı $T(N) = 2N + 3$ olur. Yani bu algoritmanın zaman karmaşıklığı $O(N)$ 'dir.

```
int lineer_arama(int A[], int n, int arama)
{
    int c; //-----> 1
    int sonuc = -1; //-----> 1

    for (c = 0; c < n; c++) //-----> N
    {
        if (A[c] == arama) //-----> N
        {
            sonuc = c;
            break;
        }
    }

    return sonuc; //-----> 1
}
```

Özyinelemeli olmayan algoritmalar için son örneğimiz matris çarpımı olacaktır. Görüldüğü üzere, matris çarpımının temel operasyonunda üç tane iç içe *for* döngüsü bulunmaktadır.

```

for (i = 0; i<N; i++)
{
    for (j = 0; j<N; j++)
    {

        C[i][j] = 0;

        for (int k = 0; k<N; k++)
        {

            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

```

Matris çarpımı için gerçekleştirilmesi gereken operasyon sayısını, toplam sembollerini kullanarak aşağıdaki gibi hesaplayabiliriz:

$$T(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \left(1 + \sum_{k=0}^{N-1} 1\right) = N^3 + N^2$$

Bu ifadeden de görüleceği üzere, matris çarpımında N^3 değeri N^2 değerinden büyük olacağı için, matris çarpımı $O(N^3)$ zaman karmaşıklığına sahiptir.

ÖZYİNELEMELİ ALGORİTMALARIN ANALİZİ

Kitabımızın önceki ünitesinde, özyinelemeli fonksiyon algoritmalarının nasıl tasarlanaçağrı işlenmişti. Bu bölümde ise özyinelemeli olarak tasarladığımız algoritmaların analizi yapılacaktır. Özyinelemeli fonksiyonların analizini yaparken gerçekleştirilecek işlemler aşağıdaki gibidir (Levitin, 2012):

1. Girdi büyüklüğünü veren parametre belirlenir.
2. Algoritmanın temel operasyonu belirlenir.
3. Girdi parametresine göre problemin temel operasyonunun çalışma sayısının değişip değizmeyeceği belirlenir.
4. Başlangıç koşulları ile birlikte algoritmanın özyinelemeli fonksiyon bağıntısı yazılır.
5. Fonksiyonların büyümesi ve toplam ifadeleri kullanılarak özyineleme bağıntısı çözüller ve zaman karmaşıklığı bulunur.

İlk örnek olarak n tane sayının toplamını özyinelemeli olarak hesaplayan fonksiyonun analizini yapalım. Bu uygulamada girdi parametresi n sayısıdır. Bu özyinelemeli fonksiyonun en küçük kısmı n sayısının 1'e eşit olduğu durumdur. Böyle olduğunda 1 sayısını geri döndürmektedir.

```

int ToplamRecursive(int n)
{
    int tmpToplam = 0;

    if (n == 1) return 1;

    tmpToplam = ToplamRecursive(n - 1);

    return tmpToplam + n;
}

```

Bu algoritmanın temel işleminin çalışma zamanı fonksiyonu aşağıdaki gibidir:

$$T(n) = \begin{cases} 1 & n=1 \\ T(n-1)+1 & n>1 \end{cases}$$

Bu ifadeyi çözecek olursak:

$$\begin{aligned} T(n) &= T(n-1)+1 \\ &= T(n-2)+1+1 \\ &= T(n-3)+1+1+1 \\ &= \dots \\ &= T(1)+1+1+\dots+1 \\ &= n*1 \\ &= n \end{aligned}$$

Buradan görüldüğü üzere n 'ye kadar olan sayıların toplamını bulan özyinelemeli fonksiyonun zaman karmaşıklığı $O(N)$ mertebesindendir.

İkinci örneğimiz faktöriyel hesabı üzerinedir. Özyinelemeli faktöriyel hesabı örneği aşağıda verilmiştir. Bu örnek için girdi n değeridir.

```
int faktoriyel(int n)
{
    if (n == 0)
        return 1;
    else
        return faktoriyel(n - 1)*n;
}
```

Çarpma işlemini gerçekleştirmek için 1 birim zaman kullanırız. Böylece, özyinelemeli çalışma zamanı ifadesini aşağıdaki gibi yazabiliriz:

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

Bu ifade, bir önceki özyineleme ifadesine çok benzerdir. Benzer şekilde çözecek olursak, faktöriyel hesabının zaman karmaşıklığı da $O(n)$ mertebesindendir.

Özyinelemeli algoritmalar için verilebilecek diğer bir örnek, ikili arama algoritmasıdır. Bu algoritmada elimizde küçükten büyüğe doğru sıralı bir dizi bulunmaktadır. Bu sıralı dizide, belirli bir elemanın olup olmadığı araştırılmaktadır. Daha önce lineer arama algoritması ile bu problemi çözmüştük. Lineer arama algoritmasının çalışma zamanı $O(n)$ 'di. İkili arama algoritmasında dizi sıralı olduğu için arama işlemini daha hızlı yapabiliriz. Bu arama algoritmasında öncelikle dizinin ortasındaki elemanı bulup aradığımız elemanı bu eleman ile karşılaştırırız. Aradığımız eleman ortadaki elemana eşit ise ortadaki elemanı cevap olarak döndürür. Aradığımız eleman ortadaki elemandan küçük ise dizi sıralı olduğundan elemanımızı dizinin sol tarafında özyinelemeli olarak aramaya devam ederiz. Benzer şekilde, aradığımız eleman ortadaki elemandan büyük ise dizinin sağ tarafında, yani daha büyük olan kısımda aramaya devam ederiz. Bu ikiye bölmeye işlemi, aranan eleman bulununcaya kadar devam eder. İkili arama algoritması aşağıda verilmiştir:

```
bool ikiliArama(int A[], int N, int eleman)
{
    int orta = N / 2;

    if (N <= 0)
        return false;

    if (key == A[orta])
        return true;
    else if (key < A[orta])
        return ikiliArama(A, orta, eleman);
    else
        return ikiliArama(&A[orta + 1], N - orta - 1, eleman);
}
```

İkili arama algoritmasının çalışma zamanını aşağıdaki gibi yazabiliriz.

$$T(N) = \begin{cases} 1 & N \leq 1 \\ T(N/2) + 1 & N > 1 \end{cases}$$

Bu özyineleme ifadesi çözüldüğünde, ikili arama algoritmasının en kötü durum çalışmasındaki zaman karmaşıklığı $O(\log N)$ şeklinde bulunur.

Özet



Algoritma analizinin önemini açıklamak

Bir problemin çözümü için farklı algoritmalar tasarlanabilir. Bu algoritmaları birbirleriyle karşılaştırmak istediğimizde genellikle çalışma zamanını ve bellek gereksinimlerini dikkate almaktayız.



Verilen bir algoritmanın analizini yapmak

Çalışma zamanı analizi yapılrken, hangi zaman biriminin dikkate alınacağı önemli bir konudur. Algoritmaları karşılaştırırken süreyi saniye, dakika vb. zaman birimi kullanarak hesaplaysak, aynı algoritmayı farklı bir bilgisayarda çalıştığımızda farklı sürede sonuca ulaşacaktır. Bu nedenle, algoritmaların çalışma zamanı hesaplanırken genellikle işlem adımı sayısından algoritmanın karmaşıklığı hesaplanarak karşılaştırma yapılmaktadır.



Algoritmaların en kötü, en iyi ve ortalama durumda verimliliklerinin farkını özetlemek

Bazı algoritmaların çözüme ulaşması girdi verisine göre farklılık göstermektedir. Girdi verisi algoritma için optimal olduğunda algoritmanın en iyi durum verimliliği, girdi verisi en kötü durumda olduğunda en kötü durum verimliliği ölçümleri yapılabilir. Olası tüm girdiler için analiz yapılp ortalaması alındığında ise ortalama durum verimliliği söz konusudur.



Asimetrik gösterimleri uygulamak

Algoritma analizinde verimliliği ölçmek için çeşitli gösterimler kullanılmaktadır. Bunlar O , Ω ve Θ gösterimleridir. Büyük O gösteriminde, verilen fonksiyonun belirli bir girdi değerinden sonra büyük O gösterimindeki fonksiyon ile üst sınırı belirtilmektedir. Büyük Ω gösteriminden daha büyük bir değere ulaşmayacağı garanti etmektedir. Büyük Ω gösterimi ise algoritmanın çalışma zamanının alt sınırını ifade etmektedir. Büyük Θ gösteriminde, belirli bir girdi değerinden daha büyük değerler için, algoritma verilen fonksiyondan kesinlikle daha büyük değerlere sahip olacaktır. Büyük Θ gösteriminde ise algoritmanın fonksiyonunu alttan ve üstten sınırlayacak bir fonksiyon bulunur.



Özyinelemeli olan ya da olmayan algoritmaların analizini yapmak

Özyinelemeli olan ya da olmayan algoritmaların analizini yaparken dikkat edilmesi gereken bazı temel adımlar vardır. Öncelikli olarak problemin girdi büyülüüğünü veren parametre belirlenir. Daha sonra algoritmanın temel operasyonu tespit edilir. Bu temel operasyonun sadece girdi parametresine bağlı olup olmadığından ve başka değerlere göre değişip değişmediğinin analizi yapılır. Bu durumlar dikkate alınarak algoritma için işlem adımlarının sayısı bulunur. Bu değer genellikle toplam ifadesiyle gösterilir. Toplam ifadeleri için verilen standart formüller ve kurallar kullanılarak algoritmanın zaman karmaşıklık ifadesi elde edilir.

Kendimizi Sınayalım

- 1.** Aşağıdakilerden hangisi özyinelemeli olmayan fonksiyonların analizindeki işlem adımlarından biri **değildir**?
- Başlangıç koşulları ile birlikte algoritmanın özyinelemeli fonksiyon bağıntısı yazılır.
 - Problemin girdi büyüklüğünü veren parametre belirlenir.
 - Temel operasyon için toplam ifadesi bulunur.
 - Algoritmanın temel operasyonu belirlenir.
 - Toplam ifadesinden çalışma zamanı bulunur.
- 2.** Aşağıdakilerin hangisinde $n = 100$ değeri için sırasıyla $\log_2 n$, $n \log_2 n$, n^2 , ve n fonksiyonlarının sonuçları küçükten büyüğe sıralanmıştır?
- $n, n \log_2 n, n^2, \log_2 n$
 - $n \log_2 n, n^2, \log_2 n, n$
 - $\log_2 n, n, n \log_2 n, n^2$
 - $n \log_2 n, \log_2 n, n, n^2$
 - $n^2, \log_2 n, n, n \log_2 n$
- 3.** Aşağıdakilerden hangisi $f(n) = n + 6$ fonksiyonu için bir üst sınır **olamaz**?
- $O(n \log n)$
 - $O(n)$
 - $O(n^2)$
 - $O(\log n)$
 - $O(n^3)$
- 4.** İki matrisi çarpma işleminin zaman karmaşıklığı aşağıdakilerden hangisidir?
- $O(n \log n)$
 - $O(n)$
 - $O(n^2)$
 - $O(\log n)$
 - $O(n^3)$
- 5.** 1'den N'ye kadar sayıların toplamını özyinelemeli olarak toplayan fonksiyon aşağıda verilmiştir.
- $$Toplam(N) = \begin{cases} 1 & N=1 \\ Toplam(N-1) + N & N>1 \end{cases}$$
- Bu özyinelemeli fonksiyon ifadesine göre, toplam fonksiyonun $N>1$ değerleri için özyinelemeli çalışma zamanı ifadesi aşağıdakilerden hangisidir?
- $T(N) = 2T(N-2) + 1$
 - $T(N) = T(N-2) + N$
 - $T(N) = T(N-1) + N$
 - $T(N) = 2T(N-1) + 1$
 - $T(N) = T(N-1) + 1$
- 6.** Bir dizi üzerinde lineer arama algoritmasında aranılan elemanın dizinin ilk elemanı olması aşağıdaki durumlardan hangisine örnektir?
- En iyi durum verimliliği
 - En kötü durum verimliliği
 - Ortalama durum verimliliği
 - Zaman verimliliği
 - Alan verimliliği
- 7.** Aşağıdaki kod parçasının zaman karmaşıklığı aşağıdakilerden hangisidir?
- ```
for (i = 0; i < n; i++)
{
 for (j = 0; j < n; j++)
 {
 for (k = 0; k < n; k++)
 m = m + i;
 }
}
```
- $O(n \log n)$
  - $O(n^3)$
  - $O(n^2)$
  - $O(\log n)$
  - $O(n)$
- 8.** Özyinelemeli olarak verilen ikili arama algoritmasının zaman karmaşıklığı aşağıdakilerden hangisidir?
- $O(n \log n)$
  - $O(n^3)$
  - $O(n^2)$
  - $O(\log n)$
  - $O(n)$
- 9.** Lineer arama algoritmasının zaman karmaşıklığı aşağıdakilerden hangisidir?
- $O(n \log n)$
  - $O(n^3)$
  - $O(n^2)$
  - $O(\log n)$
  - $O(n)$
- 10.** Aşağıdaki kod parçasının zaman karmaşıklığı aşağıdakilerden hangisidir?
- ```
for (i = 0; i < n; i++)
{
    m = m + i;
}
```
- $O(n)$
 - $O(n^3)$
 - $O(n^2)$
 - $O(\log n)$
 - $O(n \log n)$

Kendimizi Sınavalım Yanıt Anahtarı

1. a Yanınız yanlış ise “Özyinelemeli Olmayan Algoritmaların Analizi” konusunu yeniden gözden geçiriniz.
2. c Yanınız yanlış ise “Fonksiyonların Büyümesi” konusunu yeniden gözden geçiriniz.
3. d Yanınız yanlış ise “Asimptotik Gösterimler” konusunu yeniden gözden geçiriniz.
4. e Yanınız yanlış ise “Özyinelemeli Olmayan Algoritmaların Analizi” konusunu yeniden gözden geçiriniz.
5. e Yanınız yanlış ise “Özyinelemeli Algoritmaların Analizi” konusunu yeniden gözden geçiriniz.
6. a Yanınız yanlış ise “Algoritmaların En Kötü Durum, En İyi Durum ve Ortalama Durum Verimlilikleri” konusunu yeniden gözden geçiriniz.
7. b Yanınız yanlış ise “Özyinelemeli Olmayan Algoritmaların Analizi” konusunu yeniden gözden geçiriniz.
8. d Yanınız yanlış ise “Özyinelemeli Algoritma Analizi” konusunu yeniden gözden geçiriniz.
9. e Yanınız yanlış ise “Özyinelemeli Olmayan Algoritmaların Analizi” konusunu yeniden gözden geçiriniz.
10. a Yanınız yanlış ise “Özyinelemeli Olmayan Algoritmaların Analizi” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

1'den 100'e kadar olan sayıların toplamını aşağıdaki formülü kullanarak hesaplayabiliriz:

$$\sum_{i=1}^N i = \frac{N(N+1)}{2} = \frac{100(100+1)}{2} = 50 * 101 = 5050$$

Sıra Sizde 2

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
2^{10}	10	2^{10}	$10 * 2^{10}$	2^{20}	2^{30}	2^{1024}	$1024!$

Sıra Sizde 3

Bir dizideki elemanlardan en büyüğünü bulmak istediğimiz zaman, gerek en iyi gerek en kötü durumda dizideki bütün elemanları sırasıyla karşılaştırmamız gerekmektedir. Çünkü en büyük eleman hakkında bir fikrimiz yoktur. Bu nedenle, en iyi durum ve en kötü durum verimliliği birbirine eşittir ve N tane elemanın tamamı kontrol edilmelidir.

Sıra Sizde 4

$f(n)$ fonksiyonunun $\Theta(n^2)$ olduğunu gösterebilmek için aşağıdaki eşitsizliği sağlayacak n_0 , c_1 ve c_2 pozitif katsayılarını bulmamız gerekmektedir.

$$\Theta(g(n)) = \{f(n) : c_1 g(n) \leq f(n) \leq c_2 g(n), n \geq n_0\}$$

Burada $n_0 = 6$, $c_1 = 2$ ve $c_2 = 3$ alındığında aşağıdaki eşitsizlik sağlanmaktadır.

$$c_1 g(n) \leq f(n) \leq c_2 g(n), n \geq n_0$$

$$2n^2 \leq 2n^2 + n \leq 3n^2, n \geq 6$$

Göründüğü üzere $f(n) = 2n^2 + n$ fonksiyonu $\Theta(n^2)$ 'dir.

Sıra Sizde 5

Verilen kod parçasının çalışma zamanı üst sınırı $O(n^2)$ değerindedir.

Yararlanılan ve Başvurulabilecek Kaynaklar

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

Çölkesen R. (2014) **Veri Yapıları ve Algoritmalar**, 9. Baskı, Papatya Yayıncılık.

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

Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

<http://www.wikipedia.org/>

6

Amaçlarımız

Bu üniteyi tamamladıktan sonra;

- 🕒 Arama algoritması kavramını tanımlayabilecek,
- 🕒 Temel arama algoritmalarını uygulayabilecek,
- 🕒 Arama algoritmalarını birbirleriyle karşılaştırabilecek bilgi ve beceriler kazanabileceksiniz.

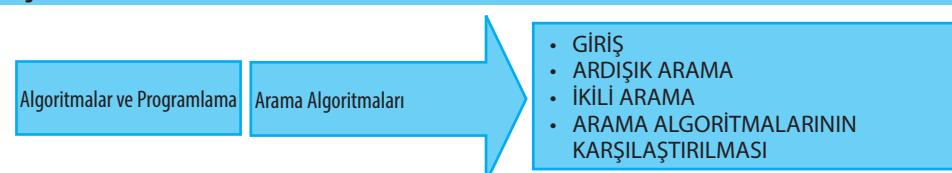
Anahtar Kavramlar

- Arama
- Ardışık Arama
- İkili Arama

İçindekiler

Algoritmalar ve Programlama

Arama Algoritmaları

- 
- GİRİŞ
 - ARDIŞIK ARAMA
 - İKİLİ ARAMA
 - ARAMA ALGORİTMALARININ KARŞILAŞTIRILMASI

Arama Algoritmaları

GİRİŞ

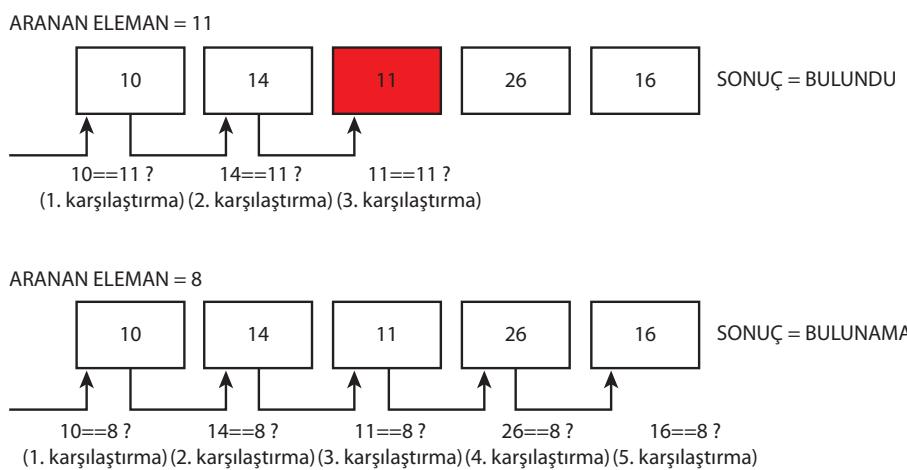
Arama, genel olarak dizilerin veya herhangi bir veri yapısının içerisinde bir elemanın bulunup bulunmadığının tespiti şeklinde ifade edilebilir. Söz konusu elemanın veri yapısı içerisinde bulunduğu tespit edilirse konum bilgisinin elde edilmesi de söz konusu olabilir. Arama algoritmaları, arama işlemini birbirlerinden farklı yöntemlerle gerçekleştirirler. Bu ünitemizde, diziler içerisindeki sayısal veriler üzerinde yapılacak aramalar hakkında bilgi verilecektir. Bu amaçla geliştirilmiş çeşitli arama algoritmaları bulunmaktadır. Temel arama algoritmaları, ardışık arama (sequential search) ve ikili arama (binary search) algoritmalarıdır. Bunların yanısıra, bu ünite kapsamında ele alınmayacak olan ve yapay zekâ problemlerinin çözümü gibi daha özel alanlarda kullanılan başka arama algoritmaları da mevcuttur. Arama algoritmalarının hepsinin kendine özel çalışma mantıkları bulunmaktadır. Örneğin, bazı algoritmaların doğru çalışması için dizilerin sıralı durumda olması gereği bilinmektedir. Sıralı durumda olmayan bir diziye bu türdeki bir arama algoritmasının uygulanması için dizinin öncelikle sıralı hale getirilmesi gereklidir. Böyle bir durumda ise sıralama ve arama algoritmalarının peş peşe çalışması söz konusu olur. Bu sebeplerle probleme uygun arama algoritmasının seçilmesi önemlidir. Ardışık arama, aranan elemanın dizinin tüm elemanlarıyla sıra sıra karşılaşılması ilkesine dayanır. Dizinin sonuna ulaşılmadan aranan elemanın bulunması söz konusu olursa algoritma sonlandırılır. Aranan elemanın bulunamadığı durumda ise bu kontroller dizinin başından sonuna kadar sürdürülür. Bu anlamda, ardışık aramanın en temel arama algoritması olduğu söylenebilir. İkili arama ise, dizinin orta noktasından alt gruplara bölünmesi ile gerçekleştirilir. Bu işlem sonrasında aranan elemanın ilgili alt grup içerisinde bulunup bulunmadığı kontrol edilir. İkili arama algoritmasının doğru çalışabilmesi için dizinin sıralı durumda olması zorunludur. Ünitemizin ilerleyen bölümlerinde, öncelikle temel arama algoritmaları örnekler yardımıyla açıklanacak ve daha sonra C program kodu ile ifade edileceklerdir.

ARDIŞIK ARAMA

Ardışık arama, en temel arama algoritmasıdır. Bu algoritmanın çalışması için dizinin sıralı olmasına ihtiyaç bulunmamaktadır. Aranan eleman, sırasıyla dizinin her bir konumdaki eleman ile karşılaştırılır. Aranan elemanın değerinin dizinin elemanlarından birisiyle aynı olduğu görülsürse algoritma başarılı bir şekilde sonlandırılır. Eğer aranan eleman dizinin içerisinde mevcut değilse, karşılaşmalar dizinin ilk elemanından son elemanına kadar sürecektr. Şekil 6.1'de ardışık arama algoritmasının çalışmasına bir örnek verilmiştir.

Şekil 6.1

Ardışık arama algoritmasının çalışmasına bir örnek



Ardışık arama algoritmasının çalışma mantığının anlatıldığı örnekte 5 elemanlı bir dizi bulunmakta olup iki farklı elemanın dizinin içerisinde aranması gerçekleştirilmiştir. Bu elemanlar sırasıyla 11 ve 8 sayılarıdır. İlk aranan 11 sayısının dizinin 3. elemani olduğu görülmektedir. Bu sebeple ilk arama 3. adımda başarı ile sonuçlandırılmış ve dizinin geride kalan elemanları kontrol edilmemiştir. Fakat daha sonra aranan 8 sayısı dizide bulunmamaktadır. Bu sebeple dizinin elemanlarının tamamının aranan eleman ile karşılaştırılması söz konusu olmaktadır. Şekilden de görüldüğü üzere 5 elemanlı bir dizi için en kötü ihtimalde 5 adet karşılaştırma yapılması gerekmektedir. Şekil 6.1'deki örneğin C program kodu ile ifade edilişi Örnek 6.1'de verilmiştir.

ÖRNEK 6.1

Ardışık arama algoritmasına yönelik C program kodu

```
/* ardisik_arama.c */

#include <stdio.h>

/* Ardisik arama fonksiyonu */
void ardisik_arama(int dizi[], int boyut, int aranan)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        if (dizi[i] == aranan)
        {
            printf("%d sayisi, dizinin %d. konumundadir.\n",
aranan, i + 1);
            break;
        }
    }
}
```

```

if (i == boyut)
    printf("%d sayisi dizide bulunamadi.\n", aranan);
}

void main()
{
    int aranan1 = 11, aranan2 = 8;
    int dizi[] = { 10, 14, 11, 26, 16 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);

    ardisik_arama(dizi, boyut, aranan1);
    ardisik_arama(dizi, boyut, aranan2);
    getch();
}

```

Program kodları, *main* fonksiyonu ile çalışmaya başlamaktadır. Daha sonra, değerleri önceden belirlenmiş olan 5 elemanlı bir dizi ve aranacak eleman *ardisik_arama* isimli C fonksiyonuna parametre olarak gönderilmektedir. Algoritma kodları, bir adet *for* döngüsü içermektedir ve bu döngü Şekil 6.1'deki karşılaştırma işlemlerinin akışını sağlamaktadır. C kodları verilen *ardisik_arama.c* uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü elde edilir.

11 sayısı, dizinin 3. konumundadır.

8 sayısı dizide bulunamadı.

ardisik_arama.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyiniz ve çalıştırarak uygun çıktıyu verdiğini kontrol ediniz.



SIRA SİZDE

1

- *aranan1* ve *aranan2* değişkenlerindeki sayılar kullanıcı tarafından girilecektir.
- *ardisik_arama* fonksiyonu, *for* döngüsü yerine *while* döngüsü kullanarak gerçekleştirilecektir.

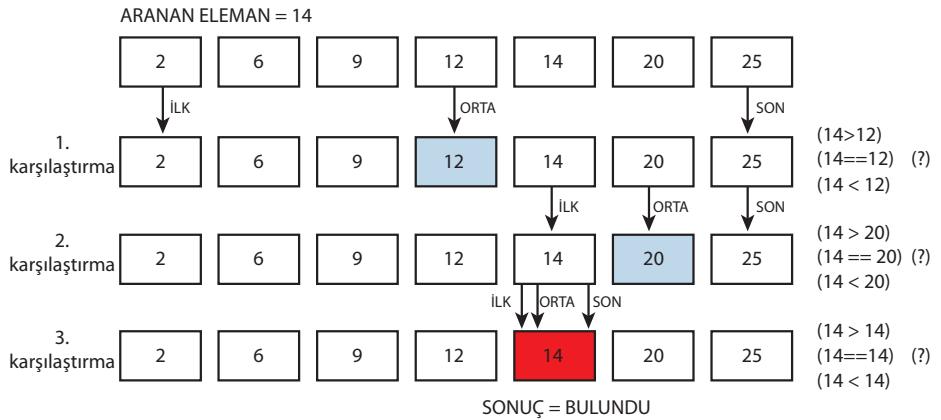
İKİLİ ARAMA

İkili arama, sıralı diziler üzerinde arama yapmak için kullanılan bir algoritmadır. Üzerinde arama yapılacak olan dizi sıralı durumda değilse, bu algoritmanın doğru çalışabilmesi için öncelikle dizinin sıralı hale getirilmesi gereklidir. İkili arama algoritması çalışmaya başladığında, ilk olarak dizinin ilk ve son elemanlarının konumları tespit edilir. Bu bilgilerden faydalılarak orta elemanın konumu hesaplanır. Daha sonra, aranan eleman dizinin orta elemanıyla karşılaştırılır ve ikisinin eşit olup olmadığına bakılır. Aranan eleman orta elemana eşit ise arama başarılı bir şekilde sonlandırılır. Aksi takdirde, aranan elemanın dizinin orta elemanından küçük veya büyük olma durumu incelenir. Aranan eleman orta elemandan büyük ise, aramaya orta eleman ile son eleman arasındaki dizi elemanları ile devam edilir. Tam tersi durum söz konusu ise, aramaya ilk eleman ile dizinin orta elemanı arasındaki dizi elemanları ile devam edilir. Bu işlem her tekrar edildiğinde ilk, orta veya son elemanların konumlarının değişmesi söz konusu olabilir. Örneğin, ilk ve son elemanın konumlarının 1 ve 3 olduğu durumda orta elemanın konumu bu iki sayının ortalaması olan 2 sayısı olacaktır. Buna karşın, ilk ve son elemanın konumlarının 4 ve 7 olduğu durumda orta elemanın konumu 5 olacaktır. Ortalama konumunun tam sayı olması zorludur. Bu örnek hesaplamada, iki sayının ortalaması olan 5.5 sayısının ondalıklı olan kısmı bu sebeple atılmakta ve bir tam sayı elde edilmektedir. Algoritmanın

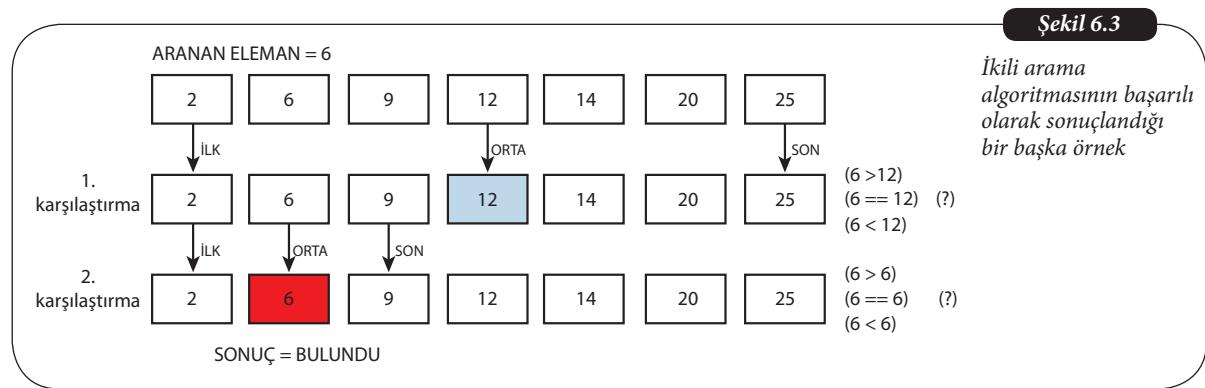
çalışması tamamlandığında aranan eleman dizide bulunamamışsa, arama işlemi başarısız bir şekilde sonlandırılır. Şekil 6.2'de ikili arama algoritmasının başarılı sonuçlandığı bir örnek verilmiştir. Bu örnek, dizinin küçükten büyüğe doğru sıralı olduğu durum için gerçekleştirilmiştir. Dizi, büyükten küçüğe doğru sıralı olursa yapılan karşıştırmalarda küçük çapta değişiklikler söz konusu olacaktır.

Sekil 6.2

İkili arama algoritmasının başarılı olarak sonuçlandığı bir örnek

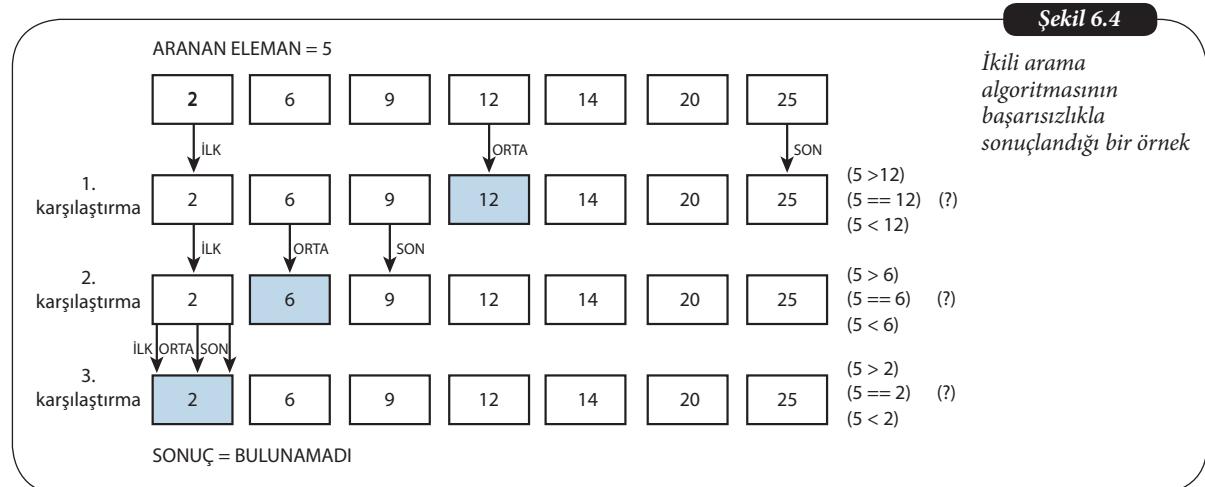


İkili arama algoritmasının çalışma mantığının anlatıldığı bu örnekte 7 elemanlı bir dizi bulunmaktadır. Söz konusu dizi içerisinde 14 sayısının aranması söz konusudur. Algoritma çalışmaya başladığında ilk, orta ve son elemanların konumları belirlenmiştir. Şekilden de görüleceği üzere dizinin 4. konumundaki 12 sayısı orta elemandır. Daha sonra, aranan eleman olan 14 sayısının orta elemana eşit, küçük veya büyük olması durumuna bakılmaktadır. Bu sayı, orta elemandan büyük olduğu için aramaya dizinin ortası ile sonu arasındaki elemanlardan devam edilir. Bu sebeple, ilk elemanın konum bilgisi güncellenmiştir ve bu güncelleme sonrasında orta elemanın konumu yeniden hesaplanmıştır. Daha sonra, aranan eleman olan 14 sayısının yeni durumdaki orta eleman olan 20 sayısına eşit, küçük veya büyük olması durumuna bakılmaktadır. 14 sayısının orta eleman olan 20'den küçük olduğu anlaşıldıından aramaya dizinin ilk ve orta elemanı arasındaki elemanlar- dan devam edilir. Bu noktada da son elemanın bilgisi, bir önceki orta eleman olan 20 sayı- sından hemen önce gelen eleman olarak güncellenmiştir. Bu durumda, ilk ve son eleman dizinin 5. konumunda yer alan 14 sayısıdır. İlk ve son elemanın konumlarının aynı olması sebebiyle tekrar orta elemanın konumu hesaplanmıştır. Bu hesaplama sonucu ilk, orta ve son elemanın konumlarının aynı duruma geldiği şekilde de görülmektedir. Orta eleman olarak belirlenen dizinin 5. konumundaki sayının 14'e eşit olması sonucu, arama işlemi başarılı bir şekilde sonlandırılmıştır. Şekilde de görüleceği üzere 14 sayısının aranması sırasında 3 adet karşılaştırma işlemi yapılmıştır. Şekil 6.3'de ikili arama algoritmasının başarılı sonuçlandığı başka bir örnek daha verilmiştir. Bu örnekte, bir önceki örneğimizdeki dizinin üzerinde 6 sayısının aranması söz konusudur.



Algoritma çalışmaya başladığında ilk, orta ve son elemanın konumları belirlenmiştir. Şekilden de görüleceği gibi dizinin 4. konumundaki 12 sayısı orta elemandır. Öncelikle, aranan eleman olan 6 sayısının orta elemana eşit, küçük veya büyük olma durumuna bakılmaktadır. Bu sayı orta elemandan küçük olduğu için aramaya dizinin ilk ve orta elemanları arasından devam edilir. Bu sebeple, öncelikle son elemanın konum bilgisi güncellenmiş ve orta elemanın konumu yeniden hesaplanmıştır. Daha sonra, aranan eleman olan 6 sayısının yeni durumdaki orta eleman olan 6 sayısına eşit, küçük veya büyük olma durumuna bakılmaktadır. İki sayı eşit olduğu için arama işlemi başarılı bir şekilde sonlandırılmış olacaktır. Şekilden de görüleceği üzere 6 sayısının aranması sırasında 2 adet karşılaştırma işlemi yapılmıştır.

Şekil 6.4'te, ikili arama algoritmasının başarısız olarak sonuçlandığı bir örnek verilmiştir. Bu örnekte, önceki örneklerimizde verilen dizinin üzerinde 5 sayısının aranması söz konusudur.



Algoritma çalışmaya başladığında ilk, orta ve son elemanın konumları belirlenmiştir. Şekilden de görüleceği gibi dizinin 4. konumundaki 12 sayısı orta elemandır. Öncelikle aranan eleman olan 5 sayısının orta elemana eşit, küçük veya büyük olma durumuna bakılmaktadır. Bu sayı orta elemandan küçük olduğu için aramaya dizinin ilk ve orta elemanları arasındaki sayılarından devam edilecektir. Bu sebeple, son elemanın konum bilgisi güncellenmiş ve orta elemanın konumu yeniden hesaplanmıştır. Daha sonra aranan eleman olan 5 sayısının yeni durumdaki orta eleman olan 6 sayısına eşit, küçük veya büyük

olma durumuna bakılmaktadır. 5 sayısının orta eleman olan 6'dan küçük olduğu anlaşılığinden aramaya dizinin ilk ve orta elemanı arasındaki elemanlardan devam edilir. Son güncelleme neticesinde ilk, orta ve son elemanın 2'ye eşit olduğu görülmektedir. Aranan eleman olan 5 sayısının 2'ye eşit olmaması sebebiyle, arama algoritması başarısız bir şekilde sonlandırılmıştır. Şekilden de görüleceği üzere 5 sayısının aranması sırasında 3 adet karşılaştırma işlemi yapılmıştır.

Şekil 6.2, 6.3 ve 6.4'teki örneklerin C program kodu ile ifade edilişi Örnek 6.2'de verilmiştir.

ÖRNEK 6.2

İkili arama algoritmasına yönelik C program kodu

```
/* ikili_arama.c */

#include <stdio.h>

/* İkili arama fonksiyonu */
void ikili_arama(int dizi[], int boyut, int aranan)
{
    int i, ilk, son, orta;
    ilk = 0;
    son = boyut - 1;
    orta = (ilk + son) / 2;
    while (ilk <= son)
    {
        printf("İlk: %d. eleman, Ortanca: %d. eleman, Son: %d. eleman\n", ilk + 1, orta + 1, son + 1);
        if (aranan > dizi[orta])
            ilk = orta + 1;
        else if (aranan == dizi[orta])
        {
            printf("%d sayisi, dizinin %d. konumunda\n", aranan, orta + 1);
            break;
        }
        else
            son = orta - 1;

        orta = (ilk + son) / 2;
    }

    if (ilk > son)
        printf("%d sayisi, dizide bulunamadi\n", aranan);
}

void main()
{
    int aranan1 = 14, aranan2 = 6, aranan3 = 5;
    int dizi[] = { 2, 6, 9, 12, 14, 20, 25 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);

    printf("İlk arama islemi:\n");
    ikili_arama(dizi, boyut, aranan1);
    printf("\nİkinci arama islemi:\n");
    ikili_arama(dizi, boyut, aranan2);
    printf("\nÜçüncü arama islemi:\n");
    ikili_arama(dizi, boyut, aranan3);
    getch();
}
```

Program, ilk olarak *main* metodundan başlamaktadır. Önceki şeillerde açıklandığı üzere, 7 elemanlı bir dizinin oluşturulduğu ve 3 farklı sayının bu dizi içerisinde *iki-li_arama* fonksiyonu yardımıyla arandığı görülmektedir. Ayrıca dizinin eleman sayısını tespit eden bir kod satırı yer almaktadır. Arama işlemi için *ikili_arama* fonksiyonuna 3 adet parametre gönderilmektedir. Bu parametreler, dizinin adresi, boyutu ve aranacak değerdir. Programda görüleceği üzere *ilk*, *orta* ve *son* değişkenlerine başlangıç değerleri atanmaktadır. Öncelikle, algoritmanın akışı gereği aranan sayının *orta* değişkeninin içeriği değerden büyük olup olmadığı kontrol edilir. Sayının, *orta* değişkenine eşit olduğu durumda, algoritma başarılı bir şekilde sonlandırılır. Sayının, *orta* değişkeninden küçük ya da büyük olması durumunda dizinin ilgili yarısı üzerinde aramalara devam edilir. Döngünün çalışması devam ettiği sürece *ilk*, *orta* ve *son* değişkenlerine yeni ifadeler atanır. Aranan sayı dizi içerisinde bulunduğu takdirde, ekrana bu sayının dizideki konumu yazdırılır. Aranan eleman bulunamadığı durumda ise bununla ilgili bir mesajın ekrana yazdırılması söz konusudur. *ikili_arama.c* program kodları çalıştırıldığında aşağıdaki ekran görüntüsü ortaya çıkmaktadır.

İlk arama işlemi:

İlk: 1. eleman, Ortanca: 4. eleman, Son: 7. eleman
 İlk: 5. eleman, Ortanca: 6. eleman, Son: 7. eleman
 İlk: 5. eleman, Ortanca: 5. eleman, Son: 5. eleman
 14 sayisi, dizinin 5. konumunda

Ikinci arama işlemi:

İlk: 1. eleman, Ortanca: 4. eleman, Son: 7. eleman
 İlk: 1. eleman, Ortanca: 2. eleman, Son: 3. eleman
 6 sayisi, dizinin 2. konumunda

Ucuncu arama işlemi:

İlk: 1. eleman, Ortanca: 4. eleman, Son: 7. eleman
 İlk: 1. eleman, Ortanca: 2. eleman, Son: 3. eleman
 İlk: 1. eleman, Ortanca: 1. eleman, Son: 1. eleman
 5 sayısı, dizide bulunamadı

Ayrıca, ikili arama algoritmasının **özyinelemeli (recursive) fonksiyon** kullanılarak gerçekleştirilmesi de mümkündür. Şekil 6.1, 6.2 ve 6.3'te verilen örneklerin özyinelemeli fonksiyon kullanılarak C program kodu ile gerçekleştirilmesi Örnek 6.3'de gösterilmektedir. Buradaki *ikili_arama_ozyinelemeli.c* program kodları çalıştırıldığında önceki ekran görüntüsünün aynısı ortaya çıkmaktadır.

Özyinelemeli (recursive) fonksiyon: Özyinelemeli fonksiyonlar, kendi içinde tekrar kendilerini çağırılan fonksiyonlardır. Bu fonksiyonlarda bir bitiş koşulu yer almaktadır. Fonksiyon, bu bitiş koşulunu sağladığında adım adım geriye değer döndürür ve sonlanır.

İkili arama algoritmasının özyinelemeli fonksiyon kullanılarak gerçekleştirilmesi

ÖRNEK 6.3

```
/* ikili_arama_ozyinelemeli.c */  

#include <stdio.h>  

/* İkili arama fonksiyonu */  

void ikili_arama_ozyinelemeli(int dizi[], int boyut, int aranan,  

int ilk, int son)  

{  

    int i, orta;
```

```

orta = (ilk + son) / 2;

if (ilk > son)
{
    printf("%d, sayisi dizide bulunamadi\n", aranan);
    return;
}

printf("Ilk: %d. eleman, Ortanca: %d. eleman, Son: %d.
eleman\n", ilk + 1, orta + 1, son + 1);

if (aranan > dizi[orta])
{
    ilk = orta + 1;
    ikili_arama_ozyinelemeli(dizi, boyut, aranan, ilk, son);
}
else if (aranan == dizi[orta])
{
    printf("%d sayisi, dizinin %d. konumunda\n", aranan,
orta + 1);
}
else
{
    son = orta - 1;
    ikili_arama_ozyinelemeli(dizi, boyut, aranan, ilk, son);
}

}

void main()
{
    int aranan1 = 14, aranan2 = 6, aranan3 = 5, ilk = 0;
    int son;
    int dizi[] = { 2, 6, 9, 12, 14, 20, 25 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);
    son = boyut - 1;

    printf("Ilk arama islemi:\n");
    ikili_arama_ozyinelemeli(dizi, boyut, aranan1, ilk, son);
    printf("\nIkinci arama islemi:\n");
    ikili_arama_ozyinelemeli(dizi, boyut, aranan2, ilk, son);
    printf("\nUcuncu arama islemi:\n");
    ikili_arama_ozyinelemeli(dizi, boyut, aranan3, ilk, son);
    getch();
}

```

Örnek 6.2'de programın bitiş koşulunun `while` döngüsü içerisinde belirtildiği görülmektedir. Özyinelemeli fonksiyon kullanılan Örnek 6.3'te ise `if` ifadesinin içerisindeki `return` anahtar kelimesi sayesinde programın sonlandırıldığı görülmektedir. Programın bu koşulu sağlamaması durumunda ise ilk ve son elemanların konumları güncellenerek, aynı `ikili_arama_ozyinelemeli` fonksiyonunun tekrar çağrıldığı görülmektedir.

ikili_arama.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyiniz ve çalıştırarak uygun çıktıyi verdığını kontrol ediniz.



SIRA SİZDE

- *ikili_arama.c* program kodundaki *main* fonksiyonu içerisindeki dizi değişkeninin değerleri { 25, 20, 14, 12, 9, 6, 2 } olarak güncellenecektir. Dolayısıyla, mevcut dizi büyükten küçüğe doğru sıralı hale getirilecektir.
- *ikili_arama* fonksiyonunun içeriğini, aynı üç aramayı doğru sonuçlandıracak şekilde düzenleyiniz.

ARAMA ALGORİTMALARININ KARŞILAŞTIRILMASI

Daha önce belirtildiği üzere, ardışık arama algoritması, içerisinde arama yapılacak olan dizinin sıralı olmasına ihtiyaç duymaz. Ancak ikili arama algoritmasının doğru çalışabilmesi için dizinin sıralı olması gereklidir. Yalnızca bu açıdan değerlendirildiğinde, ardışık arama algoritması ikili arama algoritmasına göre daha üstün görünebilir. Ancak iki algoritmanın **zaman karmaşıklığı (time complexity)** ele alındığında, ikili arama algoritmasının daha hızlı çalışabildiği görülecektir. Ardışık arama algoritmasının gerçekleşmesi sırasında, n elemanlı bir dizi için en fazla n adet karşılaştırma yapılması gerekmektedir. Dolayısıyla ardışık aramanın en kötü durumda zaman karmaşıklığının $O(n)$ olduğunu söyleyebiliriz. İkili arama algoritması ise n elemanlı bir dizi için en kötü durumda $\log_2(n)$ adet karşılaştırma yapmaya ihtiyaç duymaktadır. Dolayısıyla ikili arama algoritmasının en kötü durumda zaman karmaşıklığının $O(\log(n))$ olduğunu söyleyebiliriz. Bu karşılaştırmadan görüleceği üzere, ikili arama algoritması ardışık aramaya kıyasla çok daha hızlı çalışan bir algoritmadır. Örnek olarak, 16 elemanlı bir dizinin içerisinde arama yapılacağını ve aranan elemanın bu dizide yer almadığını varsayılmı. Bu durumda, ardışık arama 16 adet karşılaştırma işlemi yaptıktan sonra sonuca ulaşacaktır. Buna karşın, ikili arama algoritması 4 adet karşılaştırma işlemi yaparak sonuca ulaşabilir. Böyle küçük bir dizi üzerinde bile, yapılacak karşılaştırma işlem sayıları arasında büyük bir fark olduğu görülmektedir. Dolayısıyla, içerisinde arama yapılacak olan dizi sıralı ise, işlem hızı açısından değerlendirdiğinde ikili arama algoritması tercih edilmelidir.

Zaman karmaşıklığı (time complexity): Algoritmaların sonuca ulaşması için gerekli olan zaman hakkında bilgi veren bir ölçütür. En kötü durumda zaman karmaşıklığı, algoritmanın çalışmasının en uzun sürebileceği durumu ifade etmek için kullanılır.

ardisik_arama.c ve *ikili_arama.c* program kodlarındaki arama ile ilgili fonksiyonları kullanarak aşağıdaki bilgiler doğrultusunda *arama.c* isimli bir program yazınız.



SIRA SİZDE

- Program kodundaki *main* fonksiyonu içerisindeki *dizi* değişkeninin içeriği değerler, { 2, 6, 9, 12, 14, 20, 25 } olarak tanımlanacaktır. Kullanıcı, aramak istediği sayıyı klavyeden girecektir.
- Sırasıyla *ardisik_arama* ve *ikili_arama* fonksiyonları çalıştırılarak 2 defa arama yapılacak ve iki fonksiyon için yapılan karşılaştırma adetleri ayrı ayrı ekrana yazdırılacaktır.

Özet



Arama algoritması kavramını tanımlamak

Kitabımızın bu ünitesinde, arama ve arama algoritması kavramları ayrıntılı şekilde ele alınmıştır. Arama, genel olarak dizilerin veya herhangi bir veri yapısının içerisinde bir elemanın bulunup bulunmadığının tespitidır. Arama algoritmaları ise arama işlemini farklı yollarla gerçekleştiren algoritmalarıdır. Bu ünitede sayısal veriler içeren diziler üzerinde arama yapmaya yarayan algoritmaların bahsedilmiştir.



Temel arama algoritmalarını uygulamak

Bu ünitede, iki temel arama algoritması üzerinde durulmaktadır. Bunlar, ardışık arama (sequential search) ve ikili arama (binary search) algoritmalarıdır. Ardışık arama, bilinen en temel arama algoritmasıdır ve çalışma prensibi oldukça basittir. Bu algoritma, aranan elemanın dizinin her bir elemanıyla sırasıyla karşılaştırılması ilkesine dayanır. Bu esnada, aranan eleman bulunduğu takdirde algoritma başarılı bir şekilde sonlandırılır. Aksi durumda ise aranan eleman dizinin bütün elemanları ile karşılaştırılmış olur ve aranan eleman dizi içerisinde bulunamayıp arama başarısız bir şekilde sonlandırılır. Ardışık arama algoritması gerek sıralı gerek sıralı olmayan diziler üzerinde çalışabilmektedir. Buna karşılık, ikili arama algoritması sıralı diziler üzerinde çalışabilir. İkili arama algoritmasında, ilk olarak dizinin ilk ve son elemanlarının konumları tespit edilir. Bu bilgiler yardımıyla dizinin orta elemanın konumu hesaplanır. Daha sonra, aranan eleman dizinin orta elemanıyla karşılaştırılır ve ikisinin eşit olup olmadığına bakılır. Aranan eleman orta elemana eşit ise arama başarılı şekilde sonlandırılır. Böyle bir eşitlik yoksa, aranan elemanın dizinin orta elemanından küçük veya büyük olma durumu incelenir. Bu incelemenin sonucuna göre, aranan elemanın dizinin ilk ile orta bölümünü arasında veya orta ile son bölüm arasında aranacağına karar verilir. Algoritmanın sonlanıncaya kadar ilk, orta ve son elemanların konumları güncellenir. Ünitemiz içerisinde, iki temel arama algoritmasının C programlama dilinde nasıl kodlandığı örneklerle açıklanmıştır.



Arama algoritmalarını birbirleriyle karşılaştırmak

Arama algoritmaları, temel olarak zaman karmaşıklığı (time complexity) ve bazı çalışma kısıtları açısından birbirleriyle karşılaştırılmaktadır. Zaman karmaşıklığı düşük olan algoritmaların daha hızlı çalışabilme özellikleri mümkündür. Bu bölümde sadece en kötü durumdaki zaman karmaşıklıkları ele alınmıştır. Ardışık aramaının en kötü durumdaki zaman karmaşıklığının $O(n)$ olduğunu söyleyebiliriz. Ancak ikili arama için en kötü durumdaki zaman karmaşıklığı $O(\log(n))$ olmaktadır. Dolayısıyla ikili aramanın ardışık arama ya göre daha hızlı bir algoritma olduğu söylenebilir. Çalışma kısıtları açısından ise sıralı bir diziye ihtiyaç duyup duymamaları örnek verilebilir. İkili arama algoritması, ardışık aramadan farklı olarak sıralı diziler üzerinde arama yapmak için tasarlanmıştır.

Kendimizi Sınayalım

- 1.** Elemanları [7, 6, 9, 1, 14, 22] olan dizi üzerinde ardışık arama yapılarak önce 1 ve daha sonra 9 sayısının bulunup bulunmadığı kontrol edilecektir. Bu aramalar için toplam kaç karşılaştırma işlemi yapılır?
- 3
 - 4
 - 5
 - 6
 - 7
- 2.** Elemanları [12, 3, 7, 6, 9, 1, 14, 22] olan dizi üzerinde ardışık arama yapılarak önce 1 ve daha sonra 19 sayısının bulunup bulunmadığı kontrol edilecektir. Bu aramalar için toplam kaç karşılaştırma işlemi yapılır?
- 6
 - 8
 - 10
 - 12
 - 14
- I. [3, 6, 8, 17, 45, 70]
 II. [70, 45, 17, 8, 6, 3]
 III. [3, 70, 6, 45, 8, 17]
- 3.** Yukarıdaki dizilerden hangisi üzerinde ardışık arama algoritması uygulanabilir?
- Yalnızca I
 - Yalnızca III
 - I ve II
 - II ve III
 - I, II, III
- 4.** Elemanları [2, 8, 12, 22, 30, 35, 40] olan dizi üzerinde ikili arama yapılarak önce 8 ve daha sonra 35 sayısının bulunup bulunmadığı kontrol edilecektir. Bu aramalar için toplam kaç karşılaştırma işlemi yapılır?
- 2
 - 4
 - 6
 - 8
 - 10
- 5.** Elemanları [2, 8, 12, 22, 30, 35, 40] olan dizi üzerinde ikili arama yapılarak önce 22 ve daha sonra 45 sayısının bulunup bulunmadığı kontrol edilecektir. Bu aramalar için toplam kaç karşılaştırma işlemi yapılır?
- 1
 - 4
 - 6
 - 7
 - 9
- I. [3, 6, 8, 17, 45, 70]
 II. [70, 45, 17, 8, 6, 3]
 III. [3, 70, 6, 45, 8, 17]
- 6.** Yukarıdaki dizilerden hangisi üzerinde ikili arama algoritması uygulanabilir?
- Yalnızca I
 - Yalnızca III
 - I ve II
 - II ve III
 - I, II, III
- 7.** Elemanları [2, 7, 10, 13, 23, 32, 45] olan ve elemanlarının konumları 1 ile 7 arasında değişen dizi üzerinde ikili arama yapılarak 8 sayısı aranacaktır. Bu arama yapılarken 2. karşılaştırma adımında ilk, orta ve son elemanların konum bilgileri ne olur?
- İlk: 1, Orta: 4, Son: 7
 - İlk: 1, Orta: 3, Son: 7
 - İlk: 1, Orta: 2, Son: 3
 - İlk: 4, Orta: 5, Son: 6
 - İlk: 4, Orta: 5, Son: 7
- 8.** Elemanları [2, 7, 10, 13, 23, 32, 45] olan ve elemanlarının konumları 1 ile 7 arasında değişen dizi üzerinde ikili arama yapılarak 33 sayısı aranacaktır. Bu arama yapılarken 2. karşılaştırma adımında ilk, orta ve son elemanların konum bilgileri ne olur?
- İlk: 1, Orta: 2, Son: 3
 - İlk: 1, Orta: 2, Son: 4
 - İlk: 1, Orta: 4, Son: 7
 - İlk: 4, Orta: 6, Son: 7
 - İlk: 5, Orta: 6, Son: 7
- 9.** Ardışık arama algoritmasının en kötü durumdaki zaman karmaşıklığı değeri nedir?
- $O(n)$
 - $O(n^2)$
 - $O(\log(n))$
 - $n.O(\log(n))$
 - $O(1)$
- 10.** İkili arama algoritmasının en kötü durumdaki zaman karmaşıklığı değeri nedir?
- $O(\log(n))$
 - $O(n^2)$
 - $O(n)$
 - $n.O(\log(n))$
 - $O(1)$

Kendimizi Sınavalım Yanıt Anahtarı

1. e Yanınız yanlış ise “Ardışık Arama” konusunu yeniden gözden geçiriniz.
2. e Yanınız yanlış ise “Ardışık Arama” konusunu yeniden gözden geçiriniz.
3. e Yanınız yanlış ise “Ardışık Arama” konusunu yeniden gözden geçiriniz.
4. b Yanınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
5. b Yanınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
6. c Yanınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
7. c Yanınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
8. e Yanınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
9. a Yanınız yanlış ise “Arama Algoritmalarının Karşılaştırılması” konusunu yeniden gözden geçiriniz.
10. a Yanınız yanlış ise “Arama Algoritmalarının Karşılaştırılması” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

İstenilen değişiklikler yapıldıktan sonra `ardisik_arama.c` programının kodları aşağıdaki gibi olmalıdır.

```
/* ardisik_arama.c */

#include <stdio.h>

/* Ardisik arama fonksiyonu */

void ardisik_arama(int dizi[], int boyut, int aranan)
{
    int i = 0;
    while (i < boyut)
    {
        if (dizi[i] == aranan)
        {
            printf("%d sayisi, dizinin %d. konumunda\n", aranan, i + 1);
            break;
        }
        i++;
    }

    if (i == boyut)
        printf("%d sayisi dizide bulunamadi\n", aranan);
}

void main()
{
    int aranan1, aranan2;
    int dizi[] = { 10, 14, 11, 26, 16 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);
    printf("Aranan 1 = ");
    scanf("%d", &aranan1);
    printf("Aranan 2 = ");
    scanf("%d", &aranan2);

    ardisik_arama(dizi, boyut, aranan1);
    ardisik_arama(dizi, boyut, aranan2);
    getch();
}
```

Sıra Sizde 2

İstenilen değişiklikler yapıldıktan sonra ikili_arama.c programının kodları aşağıdaki gibi olmalıdır. ikili_arama fonksiyonu içerisindeki ilk if şartının “büyükür” yerine “küçüktür” şeklinde güncellenmesi programın doğru çalışmasını sağlar.

```
/* ikili_arama.c */

#include <stdio.h>

/* İkili arama fonksiyonu */
void ikili_arama(int dizi[], int boyut, int aranan)
{
    int i, ilk, son, orta;
    ilk = 0;
    son = boyut - 1;
    orta = (ilk + son) / 2;

    while (ilk <= son)
    {
        printf("İlk: %d. eleman, Ortanca: %d. eleman, Son: %d. eleman\n", ilk + 1, orta + 1, son + 1);
        if (aranan < dizi[orta])
            ilk = orta + 1;
        else if (aranan == dizi[orta])
        {
            printf("%d sayisi, dizinin %d. konumunda\n", aranan, orta + 1);
            break;
        }
        else
            son = orta - 1;

        orta = (ilk + son) / 2;
    }
    if (ilk > son)
        printf("%d, sayisi dizide bulunamadi\n", aranan);
}

void main()
{
    int aranan1 = 14, aranan2 = 6, aranan3 = 5;
    int dizi[] = { 25, 20, 14, 12, 9, 6, 2 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);

    printf("İlk arama islemi:\n");
    ikili_arama(dizi, boyut, aranan1);
    printf("\nIkinci arama islemi:\n");
    ikili_arama(dizi, boyut, aranan2);
    printf("\nUcuncu arama islemi:\n");
    ikili_arama(dizi, boyut, aranan3);
    getch();
}
```

Sıra Sizde 3

Öncelikle olarak `scanf` fonksiyonu ile kullanıcıdan, aranacak sayı bilgisi alınmalıdır. Daha sonra `ardisik_arama` ve `ikili_arama` fonksiyonları içerisine `karsilastirma_sayisi` isimli değişken eklenmiştir. Bu değişken vasıtasyyla iki arama fonksiyonunun ayrı ayrı kaç defa karşılaştırma yaptığı bilgisi ekrana yazdırılmıştır.

```

/* arama.c */
#include <stdio.h>

/* Ardisik arama fonksiyonu */
void ardisik_arama(int dizi[], int boyut, int aranan)
{
    int i, karsilastirma_sayisi = 0;
    for (i = 0; i < boyut; i++)
    {
        karsilastirma_sayisi++;
        if (dizi[i] == aranan)
        {
            printf("%d sayisi, dizinin %d. konumunda\n", aranan, i + 1);
            break;
        }
    }
    if (i == boyut)
        printf("%d sayisi dizide bulunamadi\n", aranan);
    printf("\nArdisik arama: %d adet karsilastirma yapildi\n\n", karsilastirma_sayisi);
}

/* ikili arama fonksiyonu */
void ikili_arama(int dizi[], int boyut, int aranan)
{
    int ilk, son, orta, karsilastirma_sayisi = 0;
    ilk = 0;
    son = boyut - 1;
    orta = (ilk + son) / 2;
    while (ilk <= son)
    {
        karsilastirma_sayisi++;
        printf("Ilk: %d. eleman, Ortanca: %d. eleman, Son: %d. eleman\n", ilk + 1, orta + 1, son + 1);
        if (aranan > dizi[orta])
            ilk = orta + 1;
        else if (aranan == dizi[orta])
        {
            printf("%d sayisi, dizinin %d. konumunda\n", aranan, orta + 1);
            break;
        }
        else
            son = orta - 1;
        orta = (ilk + son) / 2;
    }
    if (ilk > son)
        printf("%d, sayisi dizide bulunamadi\n", aranan);
    printf("\nIkili arama: %d adet karsilastirma yapildi\n\n", karsilastirma_sayisi);
}

void main()
{
    int aranan1;
    int dizi[] = { 2, 6, 9, 12, 14, 20, 25 };
    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);

    // Kullanici deger giriyor
    printf("Aranacak sayi = ");
    scanf("%d", &aranan1);

    // Ardisik arama yapiliyor
    ardisik_arama(dizi, boyut, aranan1);

    // Ikili arama yapiliyor
    ikili_arama(dizi, boyut, aranan1);
    getch();
}

```

Yararlanılan ve Başvurulabilecek Kaynaklar

- Isrd Group. (2007). **Data Structures using C**, Tata McGraw-Hill.
- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) **Introduction to Algorithms**, 3rd Edition, MIT Press.
- Levitin A. (2012) **Introduction to The Design & Analysis of Algorithms**, 3rd Edition, Pearson.
- Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

7

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Sıralama algoritması kavramını tanımlayabilecek,
 - 🕒 Temel sıralama algoritmalarını uygulayabilecek,
 - 🕒 Sıralama algoritmalarını birbirleriyle karşılaştırabilecek bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- Sıralama
- Temel Sıralama Algoritmaları
- Zaman Karmaşıklığı
- İstikrarlılık

İçindekiler

Algoritmalar ve Programlama

Sıralama Algoritmaları

- 
- GİRİŞ
 - BALONCUK SIRALAMASI
 - SEÇMELİ SIRALAMA
 - ARAYA SOKARAK SIRALAMA
 - HIZLI SIRALAMA
 - BİRLEŞTİREREK SIRALAMA
 - YİĞİN SIRALAMASI
 - SIRALAMA ALGORİTMALARININ ÖZELLİKLERİ

Sıralama Algoritmaları

GİRİŞ

Sıralama, genel olarak dizilerin veya herhangi bir veri yapısının elemanlarının istenilen düzene getirilmesi olarak ifade edilebilir. Bu düzenin temel olarak küçükten büyüğe veya büyükten küçüğe olması mümkündür. Sıralanmak istenilen elemanlar, sayılar veya metinsel ifadeler gibi kavramlar olabilir. Bu ünitemizde, sayısal veriler içeren dizilerin sıralanması üzerinde durulacaktır. Bu amaçla geliştirilmiş çeşitli sıralama algoritmaları bulunmaktadır. Baloncuk sıralaması (bubble sort), seçmeli sıralama (selection sort), araya sokarak sıralama (insertion sort), hızlı sıralama (quick sort), birleştirerek sıralama (merge sort), yiğin sıralaması (heap sort) gibi algoritmalar temel sıralama algoritmalarıdır. Bu algoritmaların hepsinin kendine özel çalışma mantıkları bulunmaktadır. Baloncuk sıralaması, dizinin her konumundaki elemanlarının sırasıyla sonraki konumdaki elemanlarla karşılaşılması ve gerekli durumlarda komşu elemanların yer değiştirmesine dayanır. Seçmeli sıralama, küçükten büyüğe doğru yapılacak bir sıralama işleminde her defasında dizinin en küçük elemanın tespit edilmesi ve uygun konuma yerleştirilmesine dayanır. Araya sokarak sıralama, diziden seçilen bir elemanın diğer elemanların kaydırılması vasisıyla uygun boşluğa yerleştirilmesi olarak açıklanabilir. Hızlı sıralama, dizinin genellikle orta noktasında bulunup pivot olarak adlandırılan bir elemanın seçilmesi ve kalan elemanların pivot elemandan küçük ya da büyük olmasına göre sıralanması üzerine kurulmuş bir algoritmadır. Birleştirerek sıralama, dizilerin daha küçük alt dizilere bölünmesi ve bu alt dizilerin sıralandıktan sonra birleştirilmesi şeklinde çalışan bir algoritmadır. Yiğin sıralaması, temel olarak dizinin elemanlarının bir yiğin veri yapısı üzerinde temsil edilir hale getirilmesine ve sonrasında sıralanmasına dayanır. Ünitemizin ilerleyen bölümlerde öncelikle temel sıralama algoritmaları örnekler yardımıyla detaylı olarak açıklanacak ve daha sonra C program kodu ile nasıl gerçeklendikleri gösterilecektir.

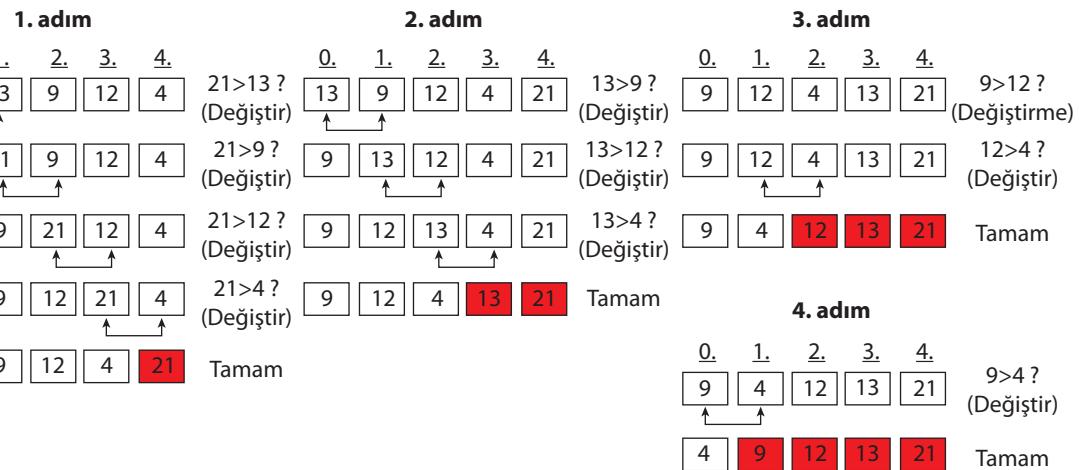
BALONCUK SIRALAMASI

Baloncuk sıralaması, temel sıralama algoritmalarından birisidir. Dizinin her bir konumundaki elemanı, sırasıyla bir sonraki konumdaki eleman ile karşılaştırılır. Dizinin küçükten büyüğe doğru sıralanması istenirse, bu karşılaştırma esnasında mevcut elemanın sonraki dizi elemanından büyük olup olmadığı kontrol edilir. Bu şartın sağlandığı sonucuna varılırsa iki eleman yer değiştirilir. Aksi durumda ise herhangi bir yer değiştirme işlemi yapılmaz ve karşılaşmaya bir sonraki konumdaki dizi elemanı ile devam edilir. Bu işlem dizinin son elemanına ulaşınca kadar sürdürülür. Dizinin başından sonuna doğru yapılan bu karşılaştırma tamamlandığında, algoritmanın bir adımı (iterasyonu) tamam-

lanmış olur. Sonuç olarak, dizinin en büyük elemanı konum olarak dizinin en sonuna yerleşmiş olacaktır. Bu algoritma adımları n elemanlı bir dizi için $n-1$ defa tekrarlandığında tamamen sıralı bir dizi elde edilmiş olunur. Şekil 7.1'de baloncuk sıralaması algoritmasının çalışmasına bir örnek verilmiştir.

Şekil 7.1

Baloncuk Sıralaması Algoritmasının Çalışmasına Bir Örnek



Baloncuk sıralamasının çalışma mantığının anlatıldığı bu örnekte 5 elemanlı bir dizi bulunmaktadır. Algoritmanın 1. adımının tamamlanması sonrasında en büyük eleman olan 21'in en sonda konumlandığı görülebilir. Her adımın bitiminde en sona konumlandırılan dizi elemanları, şekilde kırmızı ile işaretlenmiştir. Algoritmanın çalışması sırasında yapılan karşılaştırmalara da şeşin içerisinde yer verilmiştir. Algoritmanın 2. adımının sonunda, dizinin en büyük ikinci elemanı olan 13 sayısının sondan bir önceki konuma yerlesiği görülmektedir. Algoritmanın 3. adımı içerisinde, karşılaştırma işlemleri sonrasında yer değiştirme ile sonuçlanmayan bir örnek bulunmaktadır. 9 ile 12 sayısı karşılaştırılmış ve 9 sayısının 12 sayısından büyük olmaması sonucunda herhangi bir yer değiştirme yapılmamıştır. Diğer adımlar da benzer şekilde sürdürülümüş ve algoritma 4 adımda tamamlanmıştır. Sıralamanın her bir sonraki adımda yapılan karşılaştırma işlemleri sayısının azlığı görülmektedir. Bunun sebebi, her adımın bitişinde sıralanması gereken eleman sayısının azalmasıdır. Çünkü dizi elemanları sondan itibaren sıralı hale gelmeye başlamakta ve bunlar için yeniden kontrol yapılması gereklidir. Şekil 7.1'deki örneğin C program kodu ile ifade edilişi Örnek 7.1'de verilmiştir.

ÖRNEK 7.1

Baloncuk sıralama algoritmasına yönelik örnek kod

```
/* baloncuk_siralaması.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
}
```

```

        printf("\n");
    }

/* Baloncuk sıralaması fonksiyonu */
void baloncuk_siralaması(int dizi[], int boyut)
{
    int i, j, gecici;

    for (i = 0; i < (boyut - 1); i++)
    {
        for (j = 0; j < boyut - i - 1; j++)
        {
            if (dizi[j] > dizi[j + 1])
            {
                // yer degistirme işlemi
                gecici = dizi[j];
                dizi[j] = dizi[j + 1];
                dizi[j + 1] = gecici;
            }
            printf("\nAdım %d.%d: ", i + 1, j + 1);
            dizi_yazdır(dizi, boyut);
        }
    }
    printf("-----\n");
    printf("\nSıralı: ");
    dizi_yazdır(dizi, boyut);
}

void main()
{
    int aranan;
    int sırasız_dizi[] = { 21, 13, 9, 12, 4 };

    // Dizinin eleman sayısı tespit ediliyor
    int boyut = sizeof(sırasız_dizi) / sizeof(int);

    printf("\nSırasız: ");
    dizi_yazdır(sırasız_dizi, boyut);
    printf("-----\n");

    baloncuk_siralaması(sırasız_dizi, boyut);
    getch();
}

```

Program kodları, *main* fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizinin *baloncuk_siralaması* isimli C fonksiyonuna parametre olarak gönderilmesiyle devam etmektedir. Algoritma kodları, iki adet *for* döngüsü içermekte olup ilk döngü Şekil 7.1'deki adımları (iterasyonları) temsil etmektedir. İkinci döngü ise her adımda yapılan karşılaştırma işlemlerini içermektedir. *dizi_yazdır* isimli fonksiyon ise dizilerin elemanlarını ekrana yazdırmak için kullanılan yardımcı bir fonksiyondur. *baloncuk_siralaması.c* uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü ortaya çıkmaktadır. Ekran görüntüsünde, dizinin ilk halinin yanısıra işlem adımları ve dizinin sıralanmış hali yer almaktadır.

Sirasiz: 21	13	9	12	4	
<hr/>					
Adim 1.1:	13	21	9	12	4
Adim 1.2:	13	9	21	12	4
Adim 1.3:	13	9	12	21	4
Adim 1.4:	13	9	12	4	21
Adim 2.1:	9	13	12	4	21
Adim 2.2:	9	12	13	4	21
Adim 2.3:	9	12	4	13	21
Adim 3.1:	9	12	4	13	21
Adim 3.2:	9	4	12	13	21
Adim 4.1:	4	9	12	13	21
<hr/>					
Siralı: 4 9 12 13 21					

SIRA SİZDE



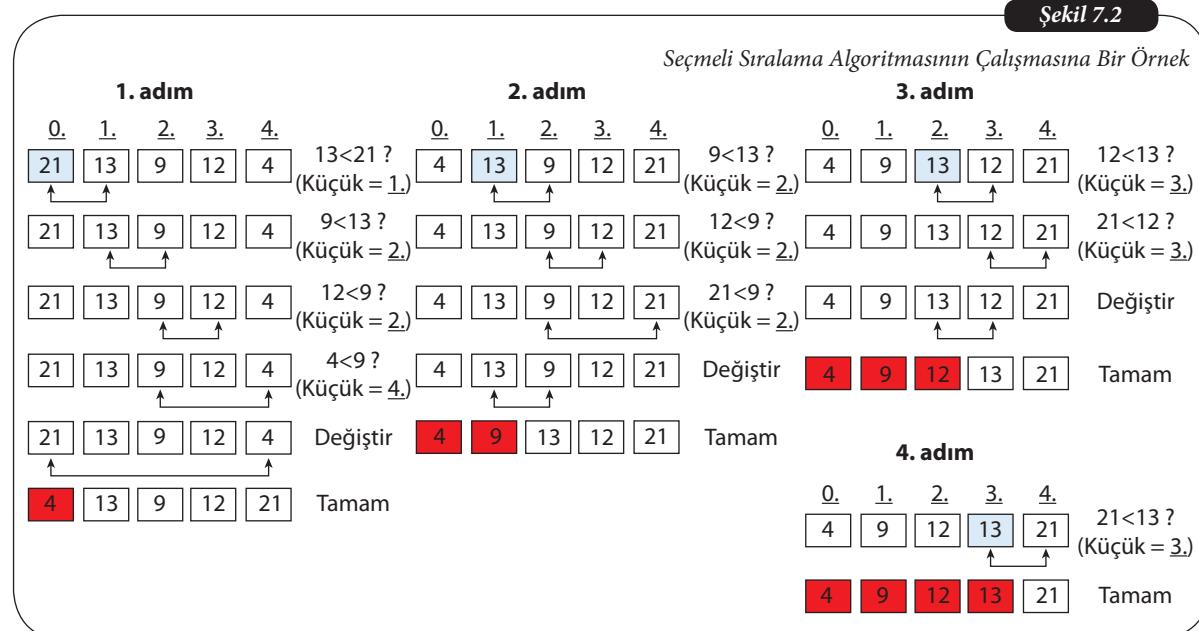
baloncuk_siralaması.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyi verdiği kontrol ediniz.

- Sıralama, küçükten büyüğe yerine büyükten küçüğe doğru yapılacaktır.
- *sirasiz_dizi* değişkeni içerisinde mevcut 5 elemanın yerine {9, 18, 12, 1} değerleri yer alacaktır.

SEÇMELİ SIRALAMA

Seçmeli sıralama, küçükten büyüğe doğru sıralama yapacağı zaman adım adımların içerisindeki en küçük elemanların bulunmasına ve bu elemanların baştan itibaren uygun konumlara yerleştirilmesine dayanan bir algoritmadır. Algoritmanın başlangıcında, dizinin ilk elemanı en küçük olarak kabul edilir ve bu eleman dizideki tüm elemanlarla tek tek karşılaştırılır. Karşılaştırmaların sonunda daha küçük bir eleman tespit edilmişse bu eleman ile ilk elemanın yerleri değiştirilir. Sonraki adımlarda ise her defasında bir sonraki elemanın dizinin kalanı için en küçük eleman olduğu varsayılar ve bu karşılaştırmalara sonraki elemanlarla devam edilir. Bu karşılaştırmalar ve yer değiştirme işlemleri dizinin bütün elemanları için tamamlandığında küçükten büyüğe doğru sıralı bir dizi elde edilir. Algoritma adımları, n elemanlı bir dizi için $n-1$ defa tekrarlandığında tamamen sıralı bir dizi elde edilir. Şekil 7.2'de seçmeli sıralama algoritmasının çalışmasına bir örnek verilmiştir.

Şekil 7.2



Seçmeli sıralamanın çalışma mantığının anlatıldığı bu örnekte 5 elemanlı bir dizi bulunmaktadır. Şekilde, her algoritma adımının başında en küçük olduğu varsayılan eleman mavi ile işaretlenmiştir. Kontroller yapılırken bu eleman kendisinden sonra gelen elemanlarla karşılaştırılmaktadır. Algoritmanın 1. adımı sonunda en küçük eleman olarak dizinin 4. konumundaki 4 değerinin tespit edildiği ve mavi ile işaretlenmiş olan 21 sayısı ile yer değiştirdiği görülebilir. Algoritmanın çalışması sırasında yapılan karşılaştırmalara da şeclin içerisinde yer verilmiştir. Her adımın bitiminde sıralı duruma gelen dizi elemanları kırmızı ile işaretlenmiştir. Algoritmanın 2. adımı sonunda, dizinin en küçük elemanı olarak 2. konumda yer alan 9 sayısı tespit edilmiş ve mavi ile işaretlenen 13 sayısı ile yerleri değiştirilmiştir. Algoritmanın 3. adımı sonunda ise 3. konumdaki 12 sayısının en küçük olarak tespit edildiği ve mavi ile işaretlenen 13 sayısı ile yer değiştirdiği görülmektedir. Son adımda ise mavi ile işaretli elemandan daha küçük bir eleman bulunmadığı için yer değiştirme işlemi yapılmamıştır. Dizinin ilk adımdından sonra yapılan karşılaştırma işlem sayısının azalmasının sebebi, her adımın bitiminde sıralanması gereken eleman sayısının azalmasıdır. Şekil 7.2'deki örneğin C program kodu ile ifade edilişi Örnek 7.2'de verilmiştir.

Seçmeli sıralama algoritmasına yönelik örnek kod

ÖRNEK 7.2

```
/* secmeli_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}
```

```

/* Secmeli siralama fonksiyonu */
void secmeli_siralama(int dizi[], int boyut)
{
    int i, j, enkucukyer, gecici;

    for (i = 0; i < boyut - 1; i++)
    {
        // ilk elemani en kucuk olarak ata
        enkucukyer = i;

        // dizinin o anki en kucuk elemanini bul
        for (j = i + 1; j < boyut; j++)
        {
            if (dizi[j] < dizi[enkucukyer])
            {
                enkucukyer = j;
            }
        }

        if (enkucukyer != i)
        {
            printf("Yer degistirdi: [ %d, %d ]\n", dizi[i],
dizi[enkucukyer]);

            // en kucuk elemani i. eleman ile degistir
            int gecici = dizi[enkucukyer];
            dizi[enkucukyer] = dizi[i];
            dizi[i] = gecici;
        }
        printf("\n%d. adim: ", i + 1);
        dizi_yazdir(dizi, boyut);
    }
    printf("-----\n");
    printf("\nSiralisi: ");
    dizi_yazdir(dizi, boyut);
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 21, 13, 9, 12, 4 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSirasiz: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    secmeli_siralama(sirasiz_dizi, boyut);
    getch();
}

```

Program kodları, `main` fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizi `secmeli_siralama` isimli C fonksiyonuna parametre olarak gönderilmektedir. Algoritma kodları, iki adet `for` döngüsü içermekte olup ilk döngü Şekil 7.2'deki adımları (iterasyonları) temsil etmektedir. İkinci döngü ise her adımda yapılan karşılaştırma işlemlerini içermektedir. `dizi_yazdir` isimli fonksiyon ise dizilerin elemanlarını yazdırmak için kullanılan yardımcı bir fonksiyondur. `secmeli_siralama.c` uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü ortaya çıkmaktadır. Ekran görüntüsünde, Şekil

7.2'de detaylı olarak açıklanmakta olan sıralama örneği için program çıktıları yer almaktadır. Dizinin ilk hali, işlem adımları ve dizinin sıralanmış hali ekrana sırasıyla yazdırılmaktadır. Yapılan yer değiştirme işlemleri de ekran görüntüsünden takip edilebilmektedir.

Sırasız: 21 13 9 12 4

Yer degistirdi: [21, 4]

1. adım: 4 13 9 12 21

Yer degistirdi: [13, 9]

2. adım: 4 9 13 12 21

Yer degistirdi: [13, 12]

3. adım: 4 9 12 13 21

4. adım: 4 9 12 13 21

Sıralı: 4 9 12 13 21

secmeli_siralama.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyi verdiğini kontrol ediniz.

- Sıralama, küçükten büyüğe yerine büyükten küçüğe doğru yapılacaktır.
- *sirasiz_dizi* değişkeni içerisinde mevcut 5 elemanın yerine {9, 18, 12, 1} değerleri yer alacaktır.



SIRA SİZDE

ARAYA SOKARAK SIRALAMA

Araya sokarak sıralama algoritması, dizinin elemanlarının kendilerinden önce gelen elemanlarla karşılaştırılması ve gerektiğiinde birbirleriyle yer değiştirmeleri prensibine dayanır. Her bir adımda (iterasyonda), dizi elemanları üzerinde soldan sağa doğru hareket ederek, kendisinden önce gelenlerle karşılaşılacak bir anahtar eleman seçilir. Bu anahtar eleman, kendisinden önce gelen diğer tüm elemanlarla sırayla karşılaşılır. Küçükten büyüğe doğru sıralama yaparken, kendisinden önce gelen eleman daha büyük ise bu eleman dizide sağa doğru kaydırılır. Dolayısıyla her adımda, dizide anahtar olarak seçilen eleman geçici olarak silinecektir. Bu karşılaştırma ve kaydırma işlemi, anahtar elemandan önceki kendişinden daha büyük olduğu sürece devam eder. Karşılaştırmalar tamamlanmışsa anahtar olarak seçilen eleman dizide uygun bir konuma yerleştirilir. Bu algoritmayı gerçek hayattan şöyle örnekleyebiliriz. Elimizde oyun kağıtları olduğunu ve içinden bir kağıt seçip çıkardığımızı varsayıyalım. Bu kağıdı kendisinden öncekilerle karşılaşır ve ondan daha büyükleri sağa doğru kaydırırız. Daha sonra, çıkardığımız ve elimizde tuttuğumuz kağıdı uygun boşluğa yerleştiririz. Dolayısıyla, o oyun kâğıdını yerleştirdiğimiz noktadan sonraki kâğıtlar birer birer sağa kaymış olacaktır. Bu algoritmda, bir anahtar eleman diziden çıkarılıp geçici olarak saklanmakta ve ondan önceki büyük elemanlar birer birer sağa kaydırılmaktadır. Algoritmanın her adımı tamamlandığında, geçici olarak diziden çıkarılan anahtar eleman uygun boşluğa yerleştirilmektedir. Şekil 7.3'te araya sokarak sıralama algoritmasının çalışmasına bir örnek verilmiştir. Bu şekilde de görüleceği üzere, her adımda yapılması gereken karşılaştırma işlemleri sayısı artmaktadır.

Şekil 7.3

Araya Sokarak Sıralama Algoritmasına Bir Örnek

1. adım

0.	1.	2.	3.	4.
21	13	9	12	4

21>13 ?
(Sağa kaydır)

2. adım

0.	1.	2.	3.	4.
13	21	9	12	4

Sağa
kaydırıldı

0.	1.	2.	3.	4.
13	21	9	12	4

0.	1.	2.	3.	4.
9	13	21	12	4

21>9 ?
(Sağa kaydır)

3. adım

0.	1.	2.	3.	4.
13	21	9	12	4

Anahtarı
yerleştir

0.	1.	2.	3.	4.
13	21	21	12	4

0.	1.	2.	3.	4.
9	13	21	21	4

Saşa
kaydırıldı

Saşa
kaydırıldı

4. adım

0.	1.	2.	3.	4.
9	12	13	21	4

21>4 ?
(Sağa kaydır)

Saşa
kaydırıldı

0.	1.	2.	3.	4.
9	12	13	21	21

13>4 ?
(Sağa kaydır)

Saşa
kaydırıldı

0.	1.	2.	3.	4.
9	12	13	13	21

13>4 ?
(Sağa kaydır)

Saşa
kaydırıldı

0.	1.	2.	3.	4.
9	12	12	13	21

9>4 ?
(Sağa kaydır)

Saşa
kaydırıldı

0.	1.	2.	3.	4.
9	9	12	13	21

Saşa
kaydırıldı

0.	1.	2.	3.	4.
4	9	12	13	21

Anahtarı
yerleştir

Araya sokarak sıralamanın çalışma mantığının anlatıldığı bu örnekte 5 elemanlı bir dizi bulunmaktadır. Her adımda bir dizi elemanı anahtar olarak belirlenmekte ve geçici olarak saklanmaktadır. Şekilde kırmızı ile işaretlenen elemanlar her adım için seçilen anahtar elemanlardır. İlk olarak dizinin 1. konumundaki 13 sayısı ile bu seçime başlanmıştır. 13 sayısı kendisinden önce gelen tek elemanın karşılaştırılmış ve önceki eleman olan 21'in sağa kaydırılmasıyla onun yerine atanmıştır. Bir sonraki adımda ise dizinin 2. konumunda yer alan 9 sayısı anahtar olarak belirlenmiş ve geçici olarak saklanmıştır. 9 sayısı, kendisinden önce gelen sayılarla karşılaştırılmış ve bu sayılarından küçük olduğu için ilk elemanın yerine atanmıştır. Algoritmanın bu şekilde 4 adım sonunda tamamlandığını ve sıralı dizinin elde edildiğini görebiliriz. Şekil 7.3'teki örneğin C program kodu ile ifade edilişi Örnek 7.3'te verilmiştir.

Araya sokarak sıralama algoritmasına yönelik örnek kod

ÖRNEK 7.3

```
/* araya_sokarak_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Araya sokarak sıralama fonksiyonu */
void araya_sokarak_siralama(int dizi[], int boyut)
{
    int i, j, anahtar;
    for (i = 1; i < boyut; i++)
    {
        anahtar = dizi[i];
        j = i - 1;
        // anahtardan büyük olanları dizide bir sağa kaydır
        while (j >= 0 && dizi[j] > anahtar)
        {
            dizi[j + 1] = dizi[j];
            j--;
        }
        dizi[j + 1] = anahtar;
        printf("\n%d. adım: ", i);
        dizi_yazdir(dizi, boyut);
    }

    printf("-----\n");
    printf("\nSıralı: ");
    dizi_yazdir(dizi, boyut);
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 21, 13, 9, 12, 4 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSırasız: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    araya_sokarak_siralama(sirasiz_dizi, boyut);
    getch();
}
```

Program kodları, *main* fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizi *araya_sokarak_siralama* isimli C fonksiyonuna para-

metre olarak gönderilmektedir. Algoritma kodları, bir adet `for` ve bir adet `while` döngüsü içermekte olup `for` döngüsü Şekil 7.4'teki adımları (iterasyonları) temsil etmektedir. `while` döngüsü ise her adımda yapılan karşılaştırma işlemlerini içermektedir. `dizi_yazdir` isimli fonksiyon ise dizilerin elemanlarını yazdırmak için kullanılan yardımcı bir fonksiyondur. `araya_sokarak_siralama.c` uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü ortaya çıkmaktadır. Bu ekran görüntüsünde, dizinin ilk hali, işlem adımları ve dizinin sıralanmış hali yer almaktadır.

Sırasız: 21 13 9 12 4

1. adım: 13 21 9 12 4

2. adım: 9 13 21 12 4

3. adım: 9 12 13 21 4

4. adım: 4 9 12 13 21

Sıralı: 4 9 12 13 21

SIRA SİZDE



3

`araya_sokarak_siralama.c` program kodlarını aşağıdaki bilgiler doğrultusunda tek-rar düzenleyip çalıştırarak uygun çıktıyi verdiğini kontrol ediniz.

- Sıralama, küçükten büyüğe yerine büyükten küçüğe doğru yapılacaktır.
- `sırasız_dizi` değişkeni içerisinde mevcut 5 elemanın yerine {9, 18, 12, 1} değerleri yer alacaktır.

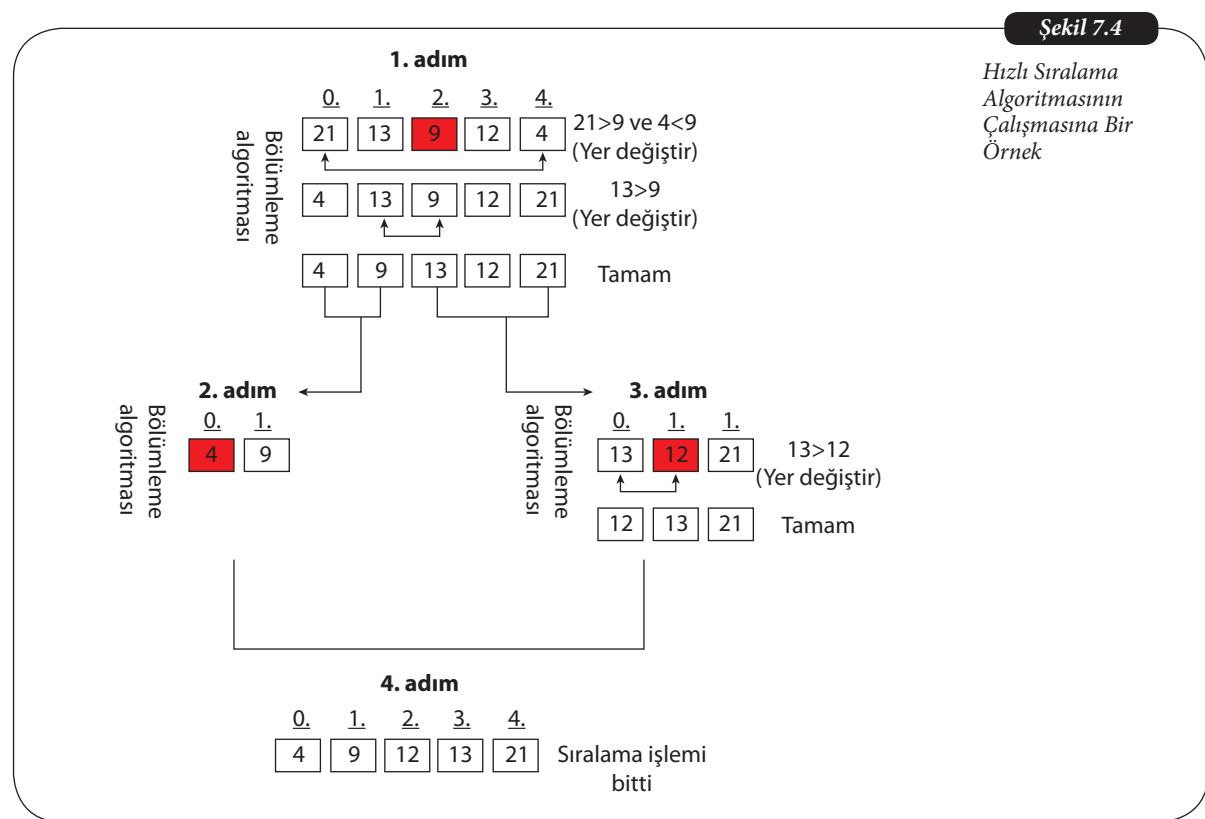
HIZLI SIRALAMA

Özyinelemeli (Recursive)

Fonksiyon: Özyinelemeli fonksiyonlar, kendi içlerinde tekrar kendilerini çağırın fonksiyonlardır. Bu fonksiyonlarda bir bitiş koşulu yer almaktadır. Fonksiyon, bu bitiş koşulunu sağladığında adım adım geriye değer döndürür ve sonlanır.

Pivot Eleman: Pivot eleman, hızlı sıralama algoritmasında bölümeleme için seçilen sınır değeridir. Hızlı sıralama algoritması, pivot elemandan küçük ve büyük değerleri iki farklı gruba ayırmaya çalışır.

Hızlı sıralama algoritması, şimdije kadar bahsedilen sıralama algoritmalarından farklı olarak böl ve yönet (divide-and-conquer) yöntemini kullanarak sıralama işlemini gerçekleştirir. Başka bir ifadeyle, diziyi mantıksal olarak farklı parçalara ayırır ve sıraladığı parçaları daha sonra birleştirir. Ayrıca, program kodunun **özyinelemeli (recursive)** fonksiyon olarak yazılması söz konusudur. Hızlı sıralama, dizinin içerisinde bir **pivot eleman** seçilmesiyle başlar. Bunun için dizinin ortasında yer alan eleman tercih edilebilir. Sonrasında pivottan küçük olduğu halde pivotun sağında yer alan elemanlar ile büyük olduğu halde pivotun solunda yer alan elemanların yerleri değiştirilir. Dolayısıyla, bu işlemin sonunda dizi, pivot elemandan küçük olanlar ve pivot elemandan büyük olanlar şeklinde iki ayrı grubu ayrılır. Daha sonra, aynı işlemler dizinin iki grubu için bağımsız olarak tekrarlanır. Algoritmanın çalışması tamamlandığında, kendi içerisinde sıralı iki dizinin birleştirilme-şile tek bir sıralı dizi elde edilir. Şekil 7.4'te hızlı sıralama algoritmasının çalışmasına bir örnek verilmiştir.



Hızlı sıralamanın çalışma mantığının anlatıldığı bu örnekte (Şekil 7.4) 5 elemanlı bir dizi bulunmaktadır. Her adımda, dizinin orta elemanı pivot olarak seçilmekte ve kırmızı ile işaretlenmektedir. Bu adımla birlikte, bölümleme algoritması çalışmaya başlar. Dizinin solundan başlanarak, pivot olarak belirlenen orta elemana kadar olan sayılar içinde pivottan büyük sayı olup olmadığı kontrol edilir. Sonrasında, dizinin sağından başlanarak pivot olarak belirlenen orta elemana kadar olan sayılar içinde pivottan küçük sayı olup olmadığı kontrol edilir. Bulunan küçük ve büyük sayılar birbirile yer değiştirilir. Şekilde, 1. adımda 21 ve 4 değerlerinin bu gerekçeyle yer değiştirdiği görülmektedir. Bu işlem, pivot olarak belirlenen sayının solundaki tüm sayıların kendisinden küçük ve sağındaki tüm sayıların kendisinden büyük hale gelmesiyle tamamlanır. Sonuç itibarıyle, pivot olan elemanın solunda ve sağında sıralı olması gerekmeyen iki adet sayı grubu yer almaktadır. Bir sonraki adımda ise dizi pivot elemanın olduğu yerden mantıksal olarak ortadan ikiye ayrılarak, iki kısım için de ayrı ayrı bölümleme algoritması uygulanır. Bu iki kısım için, kendi sayı grupları içerisinde birer pivot belirlenerek işlemler sürdürülür. En sonunda, bu iki grup da kendi içlerinde sıralı hale gelmiş olur ve iki grubun birleşmesiyle tamamen sıralı bir dizi elde edilir. Şekil 7.4'teki örneğin C program kodu ile ifade edilişi Örnek 7.4'te verilmiştir.

Hızlı sıralama algoritmasına yönelik örnek kod

ÖRNEK 7.4

```
/* hizli_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
```

```

        for (i = 0; i < boyut; i++)
        {
            printf("%d\t", dizi[i]);
        }
        printf("\n");
    }

/* Bölümleme algoritması içeren fonksiyon*/
int bolumle(int dizi[], int ilk, int son)
{
    int i = ilk;
    int j = son;
    int gecici;
    int pivot = dizi[(ilk + son) / 2];

    while (i <= j)
    {
        while (dizi[i] < pivot) i++;
        while (dizi[j] > pivot) j--;
        if (i <= j)
        {
            gecici = dizi[i];
            dizi[i] = dizi[j];
            dizi[j] = gecici;
            i++;
            j--;
        }
    }
    return i;
}

/* Hızlı sıralama fonksiyonu */
void hizli_siralama(int dizi[], int ilk, int son)
{
    int konum = bolumle(dizi, ilk, son);

    if (ilk < konum - 1)
    {
        hizli_siralama(dizi, ilk, konum - 1);
    }
    if (konum < son)
    {
        hizli_siralama(dizi, konum, son);
    }
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 21, 13, 9, 12, 4 };

    // Dizinin eleman sayısı tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSirasiz: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    hizli_siralama(sirasiz_dizi, 0, boyut - 1);
    printf("\nSiralı: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");
    getch();
}

```

Program kodları, main fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizi `hizli_siralama` isimli C fonksiyonuna parametre olarak gönderilmektedir. `hizli_siralama` isimli C fonksiyonunun, kendi içerisinde 1 defa böülüme ve 2 defa `hizli_siralama` fonksiyonunu çağrıdiği görülmektedir. Dolayısıyla, `hizli_siralama` fonksiyonu özyinelemeli bir fonksiyondur. Şekil 7.4'te de gösterildiği gibi böülüme fonksiyonu, bir pivot eleman seçtikten sonra pivot elemandan küçüklerin ve büyüklerin sırasıyla pivot elemanın soluna ve sağına geçmesini sağlamaktadır. Daha sonra 2 defa `hizli_siralama` fonksiyonu çağrılarak dizi mantıksal olarak ikiye ayrılır ve iki tarafın da kendi içerisinde tekrar böülüme fonksiyonunu çağırmasıyla dizinin sıralanması söz konusu olur. `dizi_yazdır` isimli fonksiyon ise dizilerin elemanlarını yazdırmak için kullanılan yardımcı bir fonksiyondur. `hizli_siralama.c` uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü elde edilir.

```
Sirasiz: 21 13 9 12 4
-----
Siralı: 4 9 12 13 21
-----
```

`hizli_siralama.c` program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyi verdieneni kontrol ediniz.



SIRA SİZDE

4

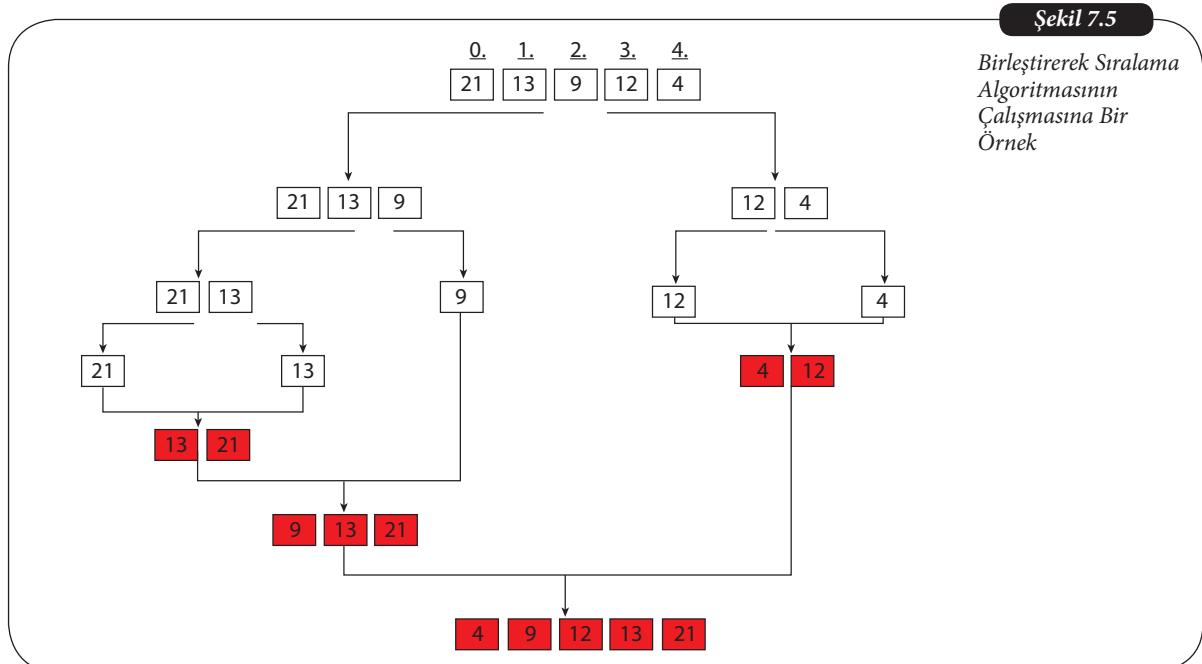
- Sıralama, küçükten büyüğe yerine büyükten küçüğe doğru yapılacaktır.
- `sirasiz_dizi` değişkeni içerisinde mevcut 5 elemanın yerine {9, 18, 12, 1} değerleri yer alacaktır.

BİRLEŞTİREREK SIRALAMA

Birleştirerek sıralama, hızlı sıralama algoritması gibi özyinelemeli bir algoritmadır. Dizi, ilk olarak orta noktadan ikiye ayrılır ve bu iki dizi kendi içinde sıralanır. Hızlı sıralamadan farklı olarak dizi içerisindeki bu iki grup oluştururken herhangi bir sayıdan küçük veya büyük şeklinde bir ayıra gidilmez. Sıralama işleminin yapılması için dizi, tek elemanlı hale gelene kadar ikiye ayrılır. Daha sonra, geçici diziler kullanılarak bu elemanlar sıralı olacak şekilde bir araya getirilirler. Şekil 7.5'te birleştirerek sıralama algoritmasının çalışmasına bir örnek verilmiştir.

Şekil 7.5

Birleştirerek Sıralama
Algoritmasının
Çalışmasına Bir
Örnek



Birleştirici sıralamanın çalışma mantığının anlatıldığı bu örnekte 5 elemanlı bir dizi bulunmaktadır. Şekilde de görüldüğü üzere dizi ilk olarak orta noktadan bölünmekte ve sırasız iki dizi elde edilmektedir. Bunun sebebi, dizinin elemanlarının mevcut sıralarıyla alt dizilere aktarılmasıdır. Söz konusu bölme işlemine, her alt dizide bir eleman kalıncaya kadar devam edilir. Daha sonra ise şekilde görüldüğü üzere kırmızı ile işaretlenen işlemlere geçilir. Bu işlemler, algoritmanın sıralı olarak birleştirme aşamasıdır. Örneğin, 21 ve 13 sayıları geçici başka bir dizi yardımıyla sıralı haliyle orijinal diziyi yazılır. Bu işlem 13, 21 ve 9 sayılarının birleştirilmesi için de sürdürülür. Burada yapılan işlem, eldeki iki dizinin sırasıyla elemanlarının kontrol edilmesi ve küçük olanın geçici diziyeye daha önce kaydedilmesidir. İşlemler, geçici diziyeye yazılmamış olan elemanların geçici diziyeye aktarılması ile sürdürülür. En sonunda tam olarak sıralanmış bir dizi, geçici diziden orijinal diziyi kopyalanır ve işlemler sona erer. Şekil 7.5'teki örneğin C program kodu ile ifade edilişi Örnek 7.5'te verilmiştir.

ÖRNEK 7.5

Birleştirerek sıralama algoritmasına yönelik örnek kod

```
/* birlestirerek_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Alt diziler sıralı olarak birleştiriliyor*/
void birlestir(int dizi[], int ilk, int orta, int son)
{
    int* gecici = (int*)malloc((son - ilk + 1)*sizeof(int));
    int i = ilk;
    int j = orta + 1;
    int k = 0;

    while (i <= orta && j <= son)
    {
        if (dizi[i] <= dizi[j])
            gecici[k++] = dizi[i++];
        else
            gecici[k++] = dizi[j++];
    }
    while (i <= orta) gecici[k++] = dizi[i++];
    while (j <= son) gecici[k++] = dizi[j++];

    k--;
    while (k >= 0)
    {
        dizi[ilk + k] = gecici[k];
        k--;
    }
    free(gecici);
}

/* Dizi, orta noktadan ikiye ayrılarak alt diziler elde ediliyor*/
```

```

void birlestirerek_siralama(int dizi[], int ilk, int son)
{
    if (ilk < son) {
        int orta = (son + ilk) / 2;
        birlestirerek_siralama(dizi, ilk, orta);
        birlestirerek_siralama(dizi, orta + 1, son);
        birlestir(dizi, ilk, orta, son);
    }
}

void main()
{
    int aranan;
    int dizi[] = { 21, 13, 9, 12, 4 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);

    printf("\nSirasiz: ");
    dizi_yazdir(dizi, boyut);
    printf("-----\n");

    birlestirerek_siralama(dizi, 0, boyut - 1);

    printf("\nSirali: ");
    dizi_yazdir(dizi, boyut);
    getch();
}

```

Program kodları, *main* fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizi *birlestirerek_siralama* isimli C fonksiyonuna parametre olarak gönderilmektedir. *birlestirerek_siralama* isimli C fonksiyonunun kendi içerisinde 2 adet *birlestirerek_siralama* ve 1 adet *birlestir* fonksiyonları çağırıldığını görmekteyiz. Dizi, *birlestirerek_siralama* fonksiyonları sayesinde orta noktadan alt dizilere ayrılacaktır. Son olarak, *birlestir* fonksiyonu sayesinde sıralı hale getirilecektir. *birlestirerek_siralama.c* uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü elde edilir.

```
Sirasiz: 21 13 9 12 4
-----
```

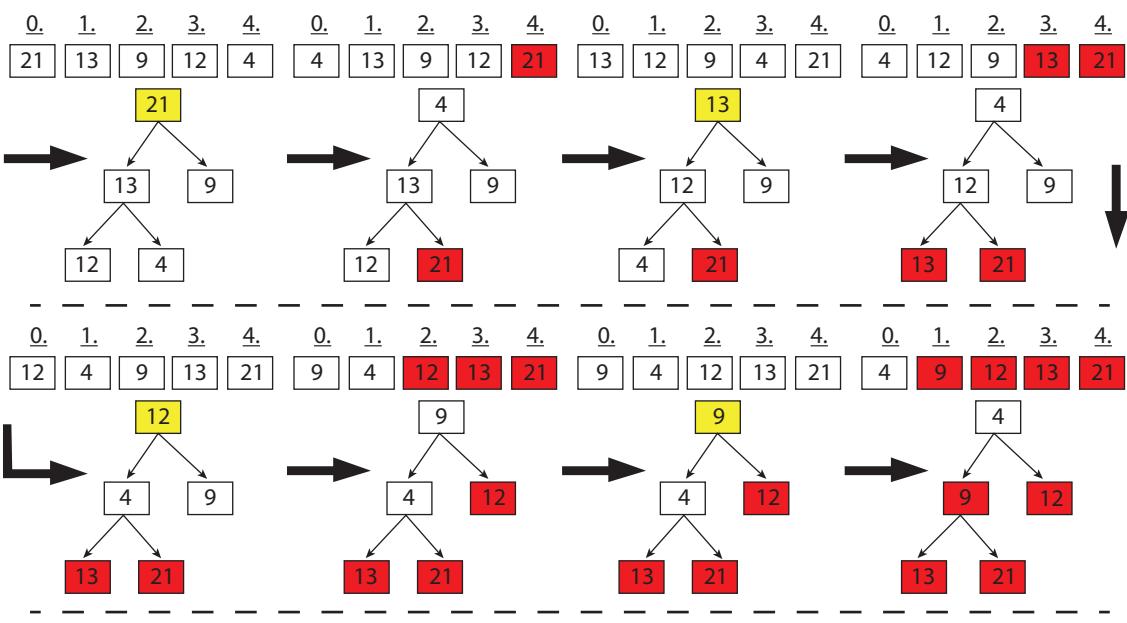
```
Sirali: 4 9 12 13 21
```

YIĞIN SIRALAMASI

Yığın sıralaması, verileri önceki ünitelerde bahsedilen yığın veri yapısı üzerinde temsil etmeye ve o yapıyı kullanarak sıralama yapmaya dayanır. Konum değerleri sıfırdan başlayan bir dizinin yığın şeklinde gösterildiği varsayıldığında, kök *i* konumunda gösteriliyorsa sol alt düğüm $2*i+1$ ve sağ alt düğüm $2*i+2$ konumlarında yer alır. Şekil 7.6'da 5 elemanlı bir dizinin yığın yapısında nasıl temsil edildiğine ve nasıl sıralandığına bir örnek verilmişdir. Örneğin, ilk dizide 0. konumdaki 21 sayısı kök iken 13 sayısı onun sol alt düğümü ve 9 sayısı onun sağ alt düğümü şeklinde gösterilir.

Şekil 7.6

Yığın Sıralaması Algoritmasının Çalışmasına Bir Örnek



Bu örnekte, her adımda yiğinin en büyük elemenini bulmakta ve sarı ile gösterilen bu elemen, ağacın kök konumuna yerleştirilmektedir. Dolayısıyla, maksimum yiğin yapısının kullanıldığı söyleyenebilir. Her bir elemenin sarı ile boyalı olarak gösterildiği aşamalarda, ağacın daha üst seviyelerindeki düğüm noktaları alttakilerden büyük veya eşit durumda olacak şekilde yer değiştirmeler yapılmıştır. Kırmızı ile gösterilen sayılar, bu kuralın kapsamında olmayan ve üzerinde artık işlem yapılmayan düğüm noktalarıdır. Bir sonraki adımda ise sarı ile gösterilen elemanlar ağaçta en alta yerleştirilmekte ve dolayısıyla dizinin son konumlarında büyük elemanların bulunması sağlanmaktadır. Her işlem adımdan sonra dizinin sonunda yer alması gereken ve kırmızı ile gösterilen eleman, sonraki sıralama işlemlerine katılmamaktadır. Şekildeki ikinci işlemde 21 sayısı en sona atıldıktan sonra, bir sonraki adımda 5 eleman yerine, kalan 4 elemanın içerisindeki en büyük sayı bulunmaktadır. İlk eleman haricindeki bütün dizi elemanları kırmızı ile işaretlendiğinde, dizi küçükten büyüğe doğru sıralı hale gelmiş olur. İlk eleman için kendisinden başka karşılaştırılacak eleman bulunmadığından algoritma bu noktada sonlandırılır. Şekil 7.6'daki örneğin C program kodu ile ifade edilişi Örnek 7.6'da verilmiştir.

ÖRNEK 7.6

Yığın sıralama algoritmasına yönelik örnek kod

```
=/* yigin_siralaması.c */

#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
```

```
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

void maksimuma_yiginla(int dizi[], int n, int i)
{
    // en buyuk elemani kok dugum olarak ata
    int enbuyuk = i;
    int sol = 2 * i + 1;
    int sag = 2 * i + 2;
    int gecici;

    // Eger sol alt agac kok dugumden buyukse
    if (sol < n && dizi[sol] > dizi[enbuyuk])
        enbuyuk = sol;

    // Eger sag alt agac kok dugumden buyukse
    if (sag < n && dizi[sag] > dizi[enbuyuk])
        enbuyuk = sag;

    // Eger enbuyuk eleman kok dugum degilse
    if (enbuyuk != i)
    {
        // en buyugu degistir
        gecici = dizi[i];
        dizi[i] = dizi[enbuyuk];
        dizi[enbuyuk] = gecici;

        // Ozyinelemeli olarak fonksiyonu cagir
        maksimuma_yiginla(dizi, n, enbuyuk);
    }
}

void yigin_siralamasi(int dizi[], int boyut)
{
    int i, gecici;
    // Yigini olustur
    for (i = boyut / 2 - 1; i >= 0; i--)
        maksimuma_yiginla(dizi, boyut, i);
    printf("\nYigin olustu: ");
    dizi_yazdir(dizi, boyut);

    // Yiginda sira sira bir eleman cikart
    for (i = boyut - 1; i > 0; i--)
    {
        printf("\n%d.Adim (Once): ", boyut - i);
        dizi_yazdir(dizi, boyut);
        // ilk konumdaki en buyuk degeri sonlara dogru kaydir
        gecici = dizi[0];
        dizi[0] = dizi[i];
        dizi[i] = gecici;
```

```

        printf("\n%d.Adim (Sonra): ", boyut - i);
        dizi_yazdir(dizi, boyut);
        maksimuma_yiginla(dizi, i, 0);
    }

}

void main()
{
    int aranan;
    int dizi[] = { 21, 13, 9, 12, 4 };

    // Dizinin eleman sayisi tespit ediliyor
    int boyut = sizeof(dizi) / sizeof(int);

    printf("\nSirasiz: ");
    dizi_yazdir(dizi, boyut);
    printf("-----\n");

    yigin_siralaması(dizi, boyut);

    printf("-----\n");
    printf("\nSirali: ");
    dizi_yazdir(dizi, boyut);
    getch();
}

```

Program kodları, *main* fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizi, *yigin_siralaması* isimli C fonksiyonuna parametre olarak gönderilmektedir. isimli C fonksiyonu, kendi içerisinde 2 adet *maksimuma_yiginla* fonksiyonunu çağrırmaktadır. İlk *maksimuma_yiginla* çağrıldığında, dizi yığın halinde temsil edilmektedir. Daha sonra *for* döngüsü içerisinde çağrılan *maksimuma_yiginla* fonksiyonu vasıtıyla, her defasında kalan elemanlar arasından en büyük olan bulunmakta ve yığının alt düğümlerine kaydırılmaktadır. Böylelikle, küçükten büyüğe doğru bir sıralama elde edilecektir. *yigin_siralama.c* uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü elde edilir.

Sirasiz:	21	13	9	12	4
<hr/>					
Yigin olustu:	21	13	9	12	4
1.Adim (Once):	21	13	9	12	4
1.Adim (Sonra):	4	13	9	12	21
2.Adim (Once):	13	12	9	4	21
2.Adim (Sonra):	4	12	9	13	21
3.Adim (Once):	12	4	9	13	21
3.Adim (Sonra):	9	4	12	13	21
4.Adim (Once):	9	4	12	13	21
4.Adim (Sonra):	4	9	12	13	21
<hr/>					
Sirali:	4	9	12	13	21

SIRALAMA ALGORİTMALARININ ÖZELLİKLERİ

Sıralama algoritmaları çeşitli açılardan birbirleriyle karşılaştırılabilirler. Bunların en önemlilerinin başında zaman karmaşıklığı (time complexity) kavramı gelir. Karmaşıklığı düşük olan bir algoritmanın daha hızlı çalışabileceği söylenebilir. Bir diğer önemli husus, algoritmanın istikrarlı (stable) olup olmamasıdır. Bir algoritmanın istikrarlı olması, dizi içerisinde aynı değere sahip elemanların bulunması durumunda, sıralama sonunda bu elemanların birbirlerine göre bağlı olarak yerlerinin değişmemesi anlamına gelmektedir. Örneğin elimizde, üzerinde sayılar bulunan kırmızı ve siyah oyun kartları olduğunu varsayıyalım. Sıralama yapmadan önce kırmızı renkli 7 sayısı, siyah renkli 7 sayısından önce gelmektedir. Sıralama yaptıktan sonra bu bağlı sıra korunuyorsa, bu algoritmanın istikrarlı olduğu söylenebilir. Ancak, yeni sırada siyah renkli 7 sayısı kırmızı renkli 7 sayısından daha önce gelmekteyse, bu durum algoritmanın istikrarlı olmadığı anlamına gelmektedir. Tablo 7.1'de sıralama algoritmaları zaman karmaşıklığı ve istikrarlılık açısından karşılaştırılmıştır. Tabloda, en kötü durumdaki zaman karmaşıklıkları dikkate alınmıştır. En kötü durum ile anlatılmak istenilen husus, algoritmanın çalışmasının en uzun süreceği durumdur. Elbette, neredeyse sıralı bir dizi verildiğinde algoritmanın sıralama işlemini daha çabuk bitirebilmesi mümkün olabilir. Tabloda da görüldüğü üzere, birleştirerek sıralama ve yiğin sıralamasının diğer sıralama algoritmalarına göre daha düşük zaman karmaşıklığına sahip olduğu anlaşılmaktadır. İstikrarlılık açısından değerlendirme yapıldığında ise yiğin sıralamasının ve seçmeli sıralamanın istikrarlı olmadığı görülmektedir. Hızlı sıralamanın genellikle istikrarlı olmaması ise algoritmanın kodlanması aşamasındaki yaklaşımlardan kaynaklanmaktadır.

Algoritma	Zaman Karmaşıklığı (En Kötü Durumda)	İstikrarlı mı?
Baloncuk sıralaması	$O(n^2)$	Evet
Seçmeli sıralama	$O(n^2)$	Hayır
Araya sokarak sıralama	$O(n^2)$	Evet
Hızlı sıralama	$O(n^2)$	Genellikle hayır
Birleştirerek sıralama	$O(n * \log(n))$	Evet
Yiğin sıralaması	$O(n * \log(n))$	Hayır

Tablo 7.1
Sıralama
Algoritmalarının
Karşılaştırılması

Özet



Sıralama algoritması kavramını tanımlamak

Kitabımızın bu ünitesinde, sıralama ve sıralama algoritması kavramlarının ne olduğu ayrıntılı bir şekilde ele alınmıştır. Sıralama, genel olarak dizilerin veya herhangi bir veri yapısının elemanlarının istenilen düzene getirilmesi olarak düşünülebilir. Sıralanması istenilen elemanların ise sayılar veya metinsel ifadeler gibi kavamlar olması mümkündür. Ünitemizde, sayısal verilerin sıralanması üzerinde durulmuştur. Sayısal veriler için bahsedilebilecek sıralama düzeni, temel olarak küçükten büyüğe veya büyükten küçüğe şeklinde olabilmektedir. Sıralama algoritmaları, sıralama işlemini kendilerine özgü çalışma mantığı sayesinde gerçekleştiren algoritmalarıdır.



Temel sıralama algoritmalarını uygulamak

Bu ünitede, altı temel sıralama algoritması üzerinde durulmaktadır. Baloncuk sıralaması (bubble sort), seçmeli sıralama (selection sort), araya sokarak sıralama (insertion sort), hızlı sıralama (quick sort), birleştirerek sıralama (merge sort), yoğun sıralaması (heap sort) başlıca sıralama algoritmalarıdır. Ünite içerisinde, sıralama algoritmalarının küçükten büyüğe doğru sıralama yaparken C programlama dilinde nasıl kodlandığı örneklerle açıklanmıştır. Hızlı sıralama, birleştirerek sıralama ve yoğun sıralaması algoritmaları, açıklanan diğer sıralama algoritmalarından farklı olarak özyinelemeli fonksiyonlar yardımıyla gerçekleştirmektedir.



Sıralama algoritmalarını birbirleriyle karşılaştırmak

Sıralama algoritmaları, temel olarak zaman karmaşıklığı (time complexity) ve istikrarlılık (stable) kavramları açısından birbirleriyle karşılaştırılmaktadır. Zaman karmaşıklığı düşük olan algoritmaların daha hızlı çalışabilmeleri mümkündür. Bu bölümde yalnızca en kötü durumda, yani algoritmaların en yavaş çalışabileceği girdiler için zaman karmaşıklıkları ele alınmıştır. Birleştirerek sıralama ve yoğun sıralaması algoritmalarının, sahip oldukları $O(n \cdot \log(n))$ karmaşılık seviyesi ile diğerlerinden daha hızlı çalışmalarıının mümkün olduğu söylenilenbilir. Çünkü baloncuk sıralaması, seçmeli sıralama, araya sokarak sıralama ve hızlı sıralama algoritmaları $O(n^2)$ karmaşılık seviyesine sahiptir. Bir algoritmanın istikrarlı olması, dizi içerisinde eşit elemanlar bulunması durumunda, sıralama sonrasında bu elemanların birbirlerine göre bağlı olarak yerlerinin değişmemesi anlamına gelir. Bu açıdan, seçmeli sıralama ve yoğun sıralaması istikrarlı değildir. Hızlı sıralama için genellikle istikrarlı değildir ifadesinin kullanılmasının sebebi ise algoritmanın kodlanma şekline göre çoğunlukla istikrarlı olmamasıdır. Bahsedilen diğer sıralama algoritmaları ise istikrarlıdır.

Kendimizi Sınayalım

- 1.** Elemanları [7, 6, 9, 1] olan bir dizi, baloncuk sıralaması algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdakilerden hangisinde doğru sırayla verilmiştir?
 - a. [1, 6, 7, 9], [9, 7, 6, 1], [1, 6, 7, 9]
 - b. [6, 7, 9, 1], [6, 7, 1, 9], [1, 6, 7, 9]
 - c. [6, 7, 1, 9], [1, 6, 7, 9], [1, 6, 7, 9]
 - d. [6, 7, 9, 1], [1, 6, 9, 7], [1, 6, 7, 9]
 - e. [6, 7, 1, 9], [6, 1, 7, 9], [1, 6, 7, 9]

 - 2.** Elemanları [5, 4, 6, 3] olan bir dizi, baloncuk sıralaması algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdakilerden hangisinde doğru sırayla verilmiştir?
 - a. [5, 3, 1, 6], [3, 5, 1, 6], [3, 4, 5, 6]
 - b. [4, 3, 5, 6], [3, 4, 5, 6], [3, 4, 5, 6]
 - c. [4, 5, 3, 6], [4, 3, 5, 6], [3, 4, 5, 6]
 - d. [5, 3, 6, 1], [5, 3, 1, 6], [3, 4, 5, 6]
 - e. [3, 5, 1, 6], [3, 1, 5, 6], [3, 4, 5, 6]

 - 3.** Elemanları [7, 1, 2, 6] olan bir dizi, seçmeli sıralama algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdakilerden hangisinde doğru sırayla verilmiştir?
 - a. [1, 7, 6, 2], [1, 2, 6, 7], [1, 2, 6, 7]
 - b. [1, 7, 2, 6], [1, 2, 7, 6], [1, 2, 6, 7]
 - c. [6, 7, 1, 2], [7, 6, 2, 1], [1, 2, 6, 7]
 - d. [2, 7, 1, 6], [2, 1, 7, 6], [1, 2, 6, 7]
 - e. [2, 7, 1, 6], [1, 2, 7, 6], [1, 2, 6, 7]

 - 4.** Elemanları [5, 6, 2, 9] olan bir dizi, seçmeli sıralama algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdakilerden hangisinde doğru sırayla verilmiştir?
 - a. [2, 6, 5, 9], [2, 5, 6, 9], [2, 5, 6, 9]
 - b. [2, 6, 5, 9], [2, 5, 9, 6], [2, 5, 6, 9]
 - c. [6, 2, 5, 9], [2, 6, 5, 9], [2, 5, 6, 9]
 - d. [9, 2, 5, 6], [2, 9, 5, 6], [2, 5, 6, 9]
 - e. [9, 2, 5, 6], [2, 5, 9, 6], [2, 5, 6, 9]

 - 5.** Elemanları [9, 8, 6, 4, 5] olan dizi, hızlı sıralama algoritması ile küçükten büyüğe doğru sıralanacaktır. 6 sayısı pivot eleman olarak seçilmiştir. Hızlı sıralama içerisindeki bölümleme algoritması 1 defa çalışıp tamamlandıktan sonra dizinin son durumu aşağıdakilerden hangisinde doğru olarak verilmiştir?
 - a. [5, 4, 6, 8, 9]
 - b. [5, 4, 6, 9, 8]
 - c. [4, 5, 6, 8, 9]
 - d. [4, 5, 6, 9, 8]
 - e. [6, 4, 5, 8, 9]

 - 6.** Elemanları [9, 4, 6, 5, 8] olan dizi, hızlı sıralaması algoritması ile büyükten küçüğe doğru sıralanacaktır. 6 sayısı pivot eleman olarak seçilmiştir. Hızlı sıralama içerisindeki bölümleme algoritması 1 defa çalışıp tamamlandıktan sonra dizinin son durumu aşağıdakilerden hangisinde doğru olarak verilmiştir?
 - a. [4, 5, 6, 8, 9]
 - b. [8, 9, 6, 4, 5]
 - c. [8, 9, 6, 5, 4]
 - d. [9, 8, 6, 4, 5]
 - e. [9, 8, 6, 5, 4]

 - 7.** Elemanları [7, 1, 2, 6] olan bir dizi, araya sokarak sıralama algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdakilerden hangisinde doğru sırayla verilmiştir?
 - a. [1, 6, 7, 2], [1, 2, 6, 7], [1, 2, 6, 7]
 - b. [2, 1, 6, 7], [1, 2, 7, 6], [1, 2, 6, 7]
 - c. [2, 1, 7, 6], [1, 2, 6, 7], [1, 2, 6, 7]
 - d. [1, 7, 2, 6], [1, 2, 7, 6], [1, 2, 6, 7]
 - e. [1, 7, 6, 2], [1, 2, 7, 6], [1, 2, 6, 7]

 - 8.** Elemanları [5, 6, 2, 9] olan bir dizi, araya sokarak sıralama algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdakilerden hangisinde doğru sırayla verilmiştir?
 - a. [5, 6, 2, 9], [2, 5, 6, 9], [2, 5, 6, 9]
 - b. [5, 6, 2, 9], [5, 6, 9, 2], [2, 5, 6, 9]
 - c. [6, 5, 2, 9], [5, 6, 2, 9], [2, 5, 6, 9]
 - d. [6, 2, 5, 9], [2, 6, 5, 9], [2, 5, 6, 9]
 - e. [9, 2, 5, 6], [2, 5, 9, 6], [2, 5, 6, 9]

 - 9.** Aşağıdaki sıralama algoritmalarından hangisinin en kötü durumda karmaşıklığı daha düşüktür?
 - a. Baloncuk sıralaması
 - b. Seçmeli sıralama
 - c. Araya sokarak sıralama
 - d. Hızlı sıralama
 - e. Birleştirerek sıralama

 - 10.**
 - I. Baloncuk sıralaması
 - II. Seçmeli sıralama
 - III. Araya sokarak sıralama
- Yukarıdaki sıralama algoritmalarından hangisi veya hangileri istikrarlıdır?
- a. I ve II
 - b. Yalnız II
 - c. I ve III
 - d. II ve III
 - e. I, II, III

Kendimizi Sınavalım Yanıt Anahtarı

1. e Yanınız yanlış ise “Baloncuk Sıralaması” konusunu yeniden gözden geçiriniz.
2. c Yanınız yanlış ise “Baloncuk Sıralaması” konusunu yeniden gözden geçiriniz.
3. b Yanınız yanlış ise “Seçmeli Sıralama” konusunu yeniden gözden geçiriniz.
4. a Yanınız yanlış ise “Seçmeli Sıralama” konusunu yeniden gözden geçiriniz.
5. a Yanınız yanlış ise “Hızlı Sıralama” konusunu yeniden gözden geçiriniz.
6. e Yanınız yanlış ise “Hızlı Sıralama” konusunu yeniden gözden geçiriniz.
7. d Yanınız yanlış ise “Araya Sokarak Sıralama” konusunu yeniden gözden geçiriniz.
8. a Yanınız yanlış ise “Araya Sokarak Sıralama” konusunu yeniden gözden geçiriniz.
9. e Yanınız yanlış ise “Sıralama Algoritmalarının Özellikleri” konusunu yeniden gözden geçiriniz.
10. c Yanınız yanlış ise “Sıralama Algoritmalarının Özellikleri” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

İstenilen değişiklikler yapıldıktan sonra *baloncuk_siralaması.c* programının kodları aşağıdaki gibi olmalıdır. *baloncuk_siralaması* fonksiyonunun içerisindeki karşılaştırma operatörünün, büyütür yerine küçütür olarak güncellenmesi gerekmektedir. Ayrıca, dizinin içerisindeki sayıların değiştirilmesi istenildiği için *sirasiz_dizi* isimli değişkenin içeriği güncellenmelidir.

```
/* baloncuk_siralaması.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Baloncuk siralaması fonksiyonu */
void baloncuk_siralaması(int dizi[], int boyut)
{
    int i, j, gecici;
    for (i = 0; i < (boyut - 1); i++)
    {
        for (j = 0; j < boyut - i - 1; j++)
        {
            if (dizi[j] < dizi[j + 1])
            {
                // yer degistirme islemi
                gecici = dizi[j];
                dizi[j] = dizi[j + 1];
                dizi[j + 1] = gecici;
            }
            printf("\nAdım %d.%d: ", i + 1, j + 1);
            dizi_yazdir(dizi, boyut);
        }
    }
    printf("-----\n");
    printf("\nSıralı: ");
    dizi_yazdir(dizi, boyut);
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 9, 18, 12, 1 };

    // Dizinin eleman sayısı tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSırasız: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    baloncuk_siralaması(sirasiz_dizi, boyut);
    getch();
}
```

Sıra Sizde 2

İstenilen değişiklikler yapıldıktan sonra *secmeli_siralama.c* programının kodları aşağıdaki gibi olmalıdır. *secmeli_siralama* fonksiyonunun içerisindeki karşılaştırma operatörünün, küçütür yerine büyütür olarak güncellenmesi gerekmektedir. Anlamsal olarak karışıklığa yol açmamak için *enkucukyer* isimli değişkenin adı *enbuyukyer* olarak değiştirilmiştir. Ayrıca, dizinin içerisindeki sayıların değiştirilmesi istenildiği için *sirasiz_dizi* isimli değişkenin içeriği güncellenmelidir.

```
/* secmeli_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Secmeli sıralama fonksiyonu */
void secmeli_siralama(int dizi[], int boyut)
{
    int i, j, enbuyukyer, gecici;

    for (i = 0; i < boyut - 1; i++)
    {
        // ilk elemanı en büyük olarak ata
        enbuyukyer = i;

        // dizinin o anki en büyük elemanını bul
        for (j = i + 1; j < boyut; j++)
        {
            if (dizi[j] > dizi[enbuyukyer])
            {
                enbuyukyer = j;
            }
        }

        if (enbuyukyer != i)
        {
            printf("Yer degistirdi: [%d, %d]\n", dizi[i], dizi[enbuyukyer]);

            // en büyük elemanı i. eleman ile değiştir
            int gecici = dizi[enbuyukyer];
            dizi[enbuyukyer] = dizi[i];
            dizi[i] = gecici;
        }
        printf("\n%d. adım: ", i + 1);
        dizi_yazdir(dizi, boyut);
    }
    printf("-----\n");
    printf("\nSıralı: ");
    dizi_yazdir(dizi, boyut);
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 9, 18, 12, 1 };

    // Dizinin eleman sayısı tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSırasız: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    secmeli_siralama(sirasiz_dizi, boyut);
    getch();
}
```

Sıra Sizde 3

İstenilen değişiklikler yapıldıktan sonra `araya_sokarak_siralama.c` programının kodları aşağıdaki gibi olmalıdır. `araya_sokarak_siralama` fonksiyonunun içerisindeki karşılaştırma operatörlerinde aşağıdaki değişikliklerin yapılması gerekmektedir. Ayrıca, dizinin içerisindeki sayıların değiştirilmesi istenildiği için `sirasiz_dizi` isimli değişkenin içeriği güncellenecektir.

```
/* araya_sokarak_siralama.c */

#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Araya sokarak sıralama fonksiyonu */
void araya_sokarak_siralama(int dizi[], int boyut)
{
    int i, j, anahtar;
    for (i = 1; i < boyut; i++)
    {
        anahtar = dizi[i];
        j = i - 1;

        // anahtardan büyük olanları dizide bir sağa kaydır
        while (j >= 0 && dizi[j] < anahtar)
        {
            dizi[j + 1] = dizi[j];
            j--;
        }
        dizi[j + 1] = anahtar;
        printf("\n%d. adım: ", i);
        dizi_yazdir(dizi, boyut);
    }

    printf("-----\n");
    printf("\nSıralı: ");
    dizi_yazdir(dizi, boyut);
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 9, 18, 12, 1 };

    // Dizinin eleman sayısı tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSırasız: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    araya_sokarak_siralama(sirasiz_dizi, boyut);
    getch();
}
```

Sıra Sizde 4

İstenilen değişiklikler yapıldıktan sonra *hizli_siralama.c* programının kodları aşağıdaki gibi olmalıdır. *bolumle* fonksiyonunun içerisindeki karşılaştırma operatörlerinde aşağıdaki değişikliklerin yapılması gerekmektedir. Ayrıca, dizinin içerisindeki sayıların değiştirilmesi istenildiği için *sirasiz_dizi* isimli değişkenin içeriği güncellenmelidir.

```
/* hizli_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdırın fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Bölümleme algoritması içeren fonksiyon*/
int bolumle(int dizi[], int ilk, int son)
{
    int i = ilk;
    int j = son;
    int gecici;
    int pivot = dizi[(ilk + son) / 2];

    while (i <= j)
    {
        while (dizi[i] > pivot) i++;
        while (dizi[j] < pivot) j--;
        if (i <= j)
        {
            gecici = dizi[i];
            dizi[i] = dizi[j];
            dizi[j] = gecici;
            i++;
            j--;
        }
    }
    return i;
}

/* Hızlı sıralama fonksiyonu */
void hizli_siralama(int dizi[], int ilk, int son)
{
    int konum = bolumle(dizi, ilk, son);

    if (ilk < konum - 1)
    {
        hizli_siralama(dizi, ilk, konum - 1);
    }
    if (konum < son)
    {
        hizli_siralama(dizi, konum, son);
    }
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 9, 18, 12, 1 };

    // Dizinin eleman sayısı tespit ediliyor
    int boyut = sizeof(sirasiz_dizi) / sizeof(int);

    printf("\nSırasız: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    hizli_siralama(sirasiz_dizi, 0, boyut - 1);
    printf("\nSıralı: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");
    getch();
}
```

Yararlanılan ve Başvurulabilecek Kaynaklar

- Cormen T.H., Leiserson C.E., Rivest R.L., Stein C. (2009) **Introduction to Algorithms**, 3rd Edition, MIT Press.
- Robert Sedgewick. (1998). **Algorithms in C**, Addison-Wesley.
- Levitin A. (2012) **Introduction to The Design & Analysis of Algorithms**, 3rd Edition, Pearson.
- Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.

8

Amaçlarımız

- Bu üniteyi tamamladıktan sonra;
- 🕒 Çizge kavramını tanımlayabilecek,
 - 🕒 Çizgelerde arama algoritmalarını uygulayabilecek,
 - 🕒 Çizgelerde en kısa yol bulma algoritmalarını kullanabilecek bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- Yönsüz Çizge
- Yönlü Çizge
- Ağırlıklandırılmış Çizge
- Enine Arama Algoritması
- Önce Derinliğine Arama Algoritması
- Dijkstra En Kısa Yol Algoritması

İçindekiler

Algoritmalar ve Programlama

Çizge Algoritmaları

- 
- GİRİŞ
 - ÇİZGELERLE İLGİLİ TEMEL KAVRAMLAR
 - ENİNE ARAMA ALGORİTMASI
 - ÖNCE DERİNLİĞİNE ARAMA ALGORİTMASI
 - DİJKSTRA EN KISA YOL ALGORİTMASI

Çizge Algoritmaları

GİRİŞ

Çizge (graph), düğümlerle bu düğümleri birbirine bağlayan kenarlardan oluşan ve ağ görünümünde olan bir tür veri yapısıdır. Temel olarak birbirleriyle ilişkili verileri temsil etmek için kullanılırlar. Çizgelerin ulaşım, bilgisayar ağları, elektrik devreleri, sosyal ağlar gibi günlük hayatın birçok alanda uygulamaları mevcuttur. Örneğin, bir sosyal ağ uygulamasını göz önüne alduğumuzda, burada kullanıcılar ve kullanıcılar arasında arkadaşlık ilişkilerinden bahsedebiliriz. Her bir kullanıcı ayrı bir düğüm noktası olarak ele alındığında, bazı düğümler diğer düğümlerle bağlantılı durumda olacaktır. Herhangi bir çizgenin içeriği düğümler, özel arama algoritmalarıyla bir noktadan başlanarak belirli sırada ziyaret edilebilmekte ve listelenebilmektedir. Çizgeler üzerinde bu işlemin yapılabilmesi için temel olarak enine arama (breadth-first search) ve önce derinliğine arama (depth-first search) algoritmaları mevcuttur. Bunun yanısıra, çizgelerin içeriği düğüm noktaları arasındaki en kısa mesafelerin hesaplanması da söz konusu olabilir. Örneğin, bir şehirden başka bir şehire araçla en hızlı şekilde ulaşmak istenildiğini varsayıyalım. Bu durumda, ilgili haritadaki tüm şehirler birer düğüm noktası olur ve belli düğüm noktaları arasında bağlantılar bulunur. Bu konuda uygun çözümlere ulaşmak için çeşitli “en kısa yol bulma” algoritmaları mevcuttur. Ünitemizde, bu tipte en yaygın algoritmaların birisi olan Dijkstra en kısa yol algoritması anlatılmaktadır. Bu algoritma, çizgenin içerisindeki herhangi bir düğüm noktasından diğer tüm düğüm noktalarına olan en kısa mesafeleri hesaplamak ve bu mesafelerin elde edilmesi için izlenecek yolları belirlemek için kullanılır. Ünitemizin ilerleyen bölümlerinde, öncelikle çizgelerle ilgili temel kavramlar ve şu ana kadar bahsedilen algoritmalar örnekler yardımıyla detaylı olarak açıklanacaktır. Ayrıca, ilgili bölümlerde algoritmaların C program kodu ile nasıl ifade edildikleri gösterilecektir.

ÇİZGELERLE İLGİLİ TEMEL KAVRAMLAR

Çizgeler, matematiksel olarak aşağıdaki formüldeki gibi temsil edilirler. Bu formülde, \mathcal{C} ifadesi çizgeyi, D ve K ifadeleri ise sırasıyla düğümleri ve kenar bağlantılarını temsil etmektedir.

$$\mathcal{C} = (D, K)$$

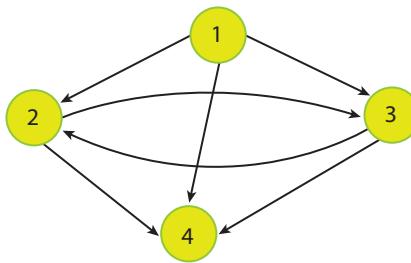
Çizgeler, düğümler arasındaki kenar bağlantılarının tipine göre **yönlü çizge (directed graph)** ve **yönsüz çizge (undirected graph)** olmak üzere ikiye ayrılırlar. Şekil 8.1'de basit bir yönlü çizge örneği bulunmaktadır.

Yönlü çizge (directed graph):
Yönlü çizge, kenar bağlantılarının yönleri temsil eden oklarla gösterildiği çizgedir. Çizge içerisindeki birbirine bağlı iki düğüm noktası arasında, sadece ilgili okun işaret ettiği yönde ilerlenebilmesi mümkündür.

Yönsüz çizge (undirected graph): Yönsüz çizge, kenar bağlantılarının yönleri temsil eden oklar ile gösterilmemiş çift yönlü olan çizgedir. Çizge içerisindeki birbirine bağlı iki düğüm noktası arasında her iki yönde de ilerlenebilmesi mümkündür. Bir anlamda, düğümler arasındaki bağlantıların simetrik olduğu söylenebilir.

Sekil 8.1

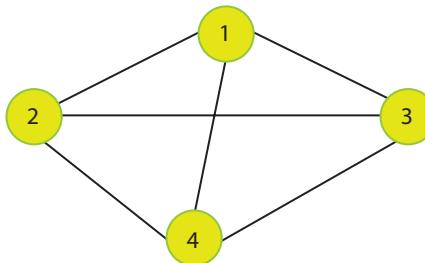
Yönlü Çizge Örneği



Şekil 8.1'deki içerik, $D=\{1, 2, 3, 4\}$ ve $K=\{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 2), (3, 4)\}$ şeklinde gösterilebilir. D kümesinin içerisinde, şekildeki gibi 4 farklı düğüm noktası bulunmaktadır. K ise şekildeki 7 adet kenar bağlantılarını temsil etmektedir. Şekildeki oklar bu çizgenin yönlü bir çizge olduğunu gösterir. Örneğin, K içerisindeki $(1,2)$ ifadesi, 1 düğümü ile 2 düğümü arasında bir kenar bağlantısı olduğunu göstermektedir. Şekil 8.2'de ise basit bir yönsüz çizge örneği bulunmaktadır.

Sekil 8.2

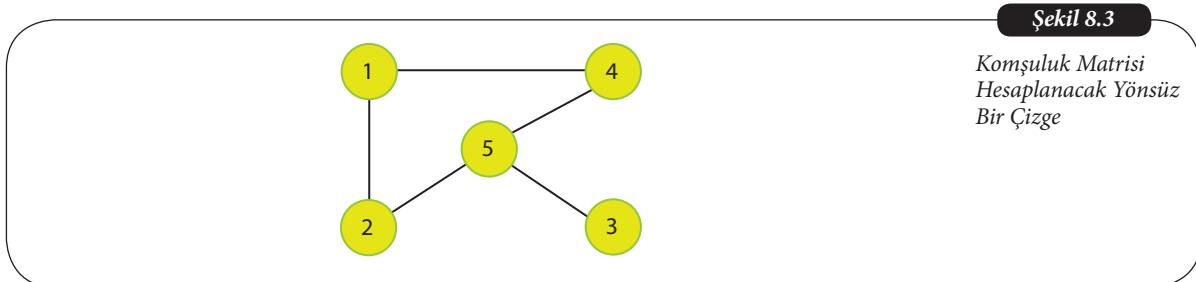
Yönsüz Çizge Örneği



Şekil 8.2'de, kenar bağlantıları oklar kullanılmadan temsil edilmektedir. Dolayısıyla, bu çizgenin yönsüz bir çizge olduğu anlaşılabılır. Şekildeki içerik, küme gösterimleri kullanılarak $D=\{1, 2, 3, 4\}$ ve $K=\{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 2), (3, 4)\}$ şeklinde ifade edilebilir. D kümesinin içerisinde 4 farklı düğüm noktası bulunmaktadır. K ise şekildeki 6 adet kenar bağlantılarını temsil etmektedir. Örneğin, K içerisindeki $(1,2)$ ifadesi, 1 düğümünden 2 düğümü arasında bir kenar bağlantısı olduğunu göstermektedir. Yönsüz çizgenin küme ile gösterimi sırasında K içerisinde $(2,3)$ bağlantısı var ise $(3,2)$ bağlantısının olmasına gerek yoktur. Bunlardan birisinin yer alması, diğerinin de var olduğunu gösterecektir. Çizgelerde komşuluk (adjacency) ve çakışım (incidence) olmak üzere iki farklı ilişki türü vardır. Komşuluk, iki düğüm arasında bir bağlantı olmasıdır. Az önceki örneği düşünecek olursak, K kümesi içerisinde $(2,3)$ bağlantısının olması, 3 düğümünün 2 düğümüne komşu olduğunu gösterir. Bir çizgenin tüm düğümleri birbirine komşu ise o çizgenin tam çizge (complete graph) olduğunu söyleyebiliriz. Çakışım ise düğüm ile kenar bağlantısı arasındaki ilişkiyi belirtir. Örneğin, K kümesi içerisinde $(2,3)$ bağlantısının, 2'den çıkan ve 3'e giren bir çakışım olduğunu söyleyebiliriz.

Çizgelere özgü yol, bağlantılılık gibi bazı kavramlar bulunmaktadır. Yol (path), çizgenin içerisinde bir düğümden başka bir düğüme ulaşmak için geçilmesi gereken düğümlerdir. Bir yolda tekrar edilen düğümler yoksa bu yola, basit yol (simple path) denilir. Çizgelerlarındaki bir diğer önemli husus ise düğümlerin komşuluk ilişkilerinin temsil edilme şeklidir. Bu amaçla kullanılabilen en çok bilinen yöntemlerden birisi komşuluk

matrisi (adjacency matrix) yöntemidir. Şekil 8.3'te komşuluk matrisi hesaplanacak olan yönsüz bir çizge bulunmaktadır. Komşuluk matrisi, aynı zamanda çizge algoritmalarının programlama yoluyla bilgisayar ortamında ifade edilmesi amacıyla da kullanılmaktadır.



Tablo 8.1'de, Şekil 8.3'teki yönsüz çizgeye ait komşuluk matrisi gösterilmektedir. Bu tabloda, aralarında komşuluk olan düğümler için 1 değeri kullanılırken, komşuluk olmayan düğümler için 0 değeri kullanılmaktadır. İlk satırda iki adet değerin 1 olmasının sebebi, 1 düşümünün 2 ve 4 düğümleri ile komşu olmasıdır. Başka bir komşuluk olmadığı için ilk satırındaki diğer değerler 0 ile gösterilmektedir. Son satırda ise üç adet değerin 1 olmasının sebebi, 5 düşümünün 2, 3 ve 4 düğümleri ile komşu olmasıdır. Diğer düğümlerle 5 düşümü arasında herhangi bir komşuluk ilişkisi olmadığı için diğerleri 0 ile gösterilmektedir. Tablo 8.1, Şekil 8.3'teki çizge görselinin matematiksel olarak ifade edilmiş halidir.

Düğüm	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

Tablo 8.1
*Komşuluk Matrisi
Örneği*

Tablo 8.1'deki matriste, düğümler arasında kenar bağlantılarının bulunup bulunmadığını göstermek için 0 ve 1 değerleri kullanılmıştır. Bu örnekte, çizgenin kenar bağlantıları sayısal olarak ağırlıklandırılmıştır ve bu yüzden gösterim için sadece 1 değeri kullanılabilir. Fakat **ağırlıklandırılmış çizge** (**weighted graph**) kavramından söz edilirse, bu değerlerin 1 yerine farklı sayılar olması da söz konusu olabilir.

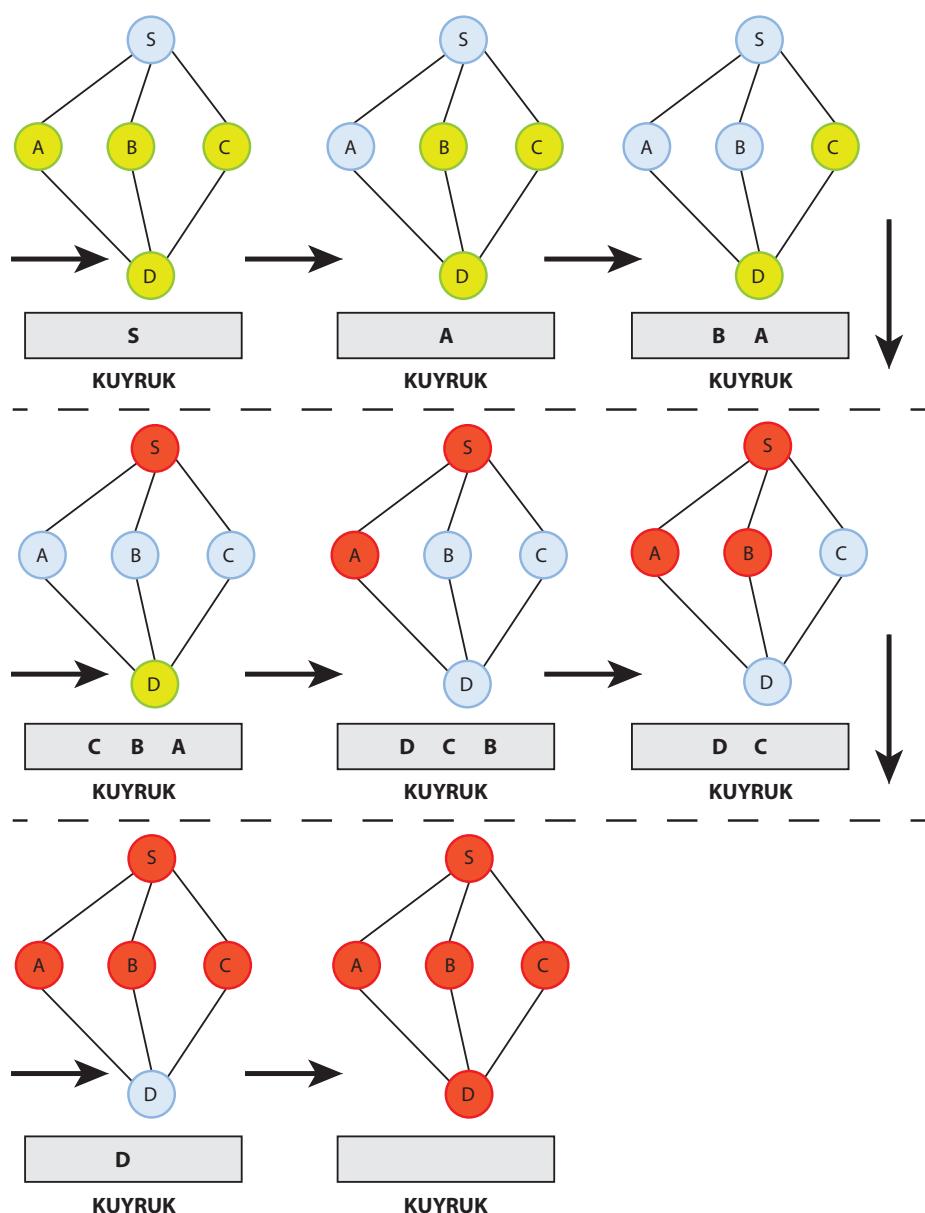
Ağırlıklandırılmış çizge (**weighted graph**): Düğümler arasındaki kenar bağlantıları üzerinde sıfırdan farklı sayısal değerlerin yer aldığı çizge türüdür.

ENİNE ARAMA ALGORİTMASI

Enine arama, çizgenin bir düşümünden başlanarak, söz konusu düşümün komşu düğümlerinin ve onların da komşularının sırayla ziyaret edildiği arama algoritmasıdır. Bu algoritmanın çalışması sırasında, öncelikle başlangıç düşümünün tüm komşuları ziyaret edilir. Daha sonra, başlangıç düşümünün komşuları ile komşu olan düğümlerden devam edilir. Algoritmanın uygulanması esnasında kuyruk (queue) veri yapısından faydalанılır. Şekil 8.4'te enine arama algoritmasının çalışmasına bir örnek verilmiştir. Bu örnekte, bir düşümün birden fazla komşusu olduğunda bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilmektedir.

Şekil 8.4

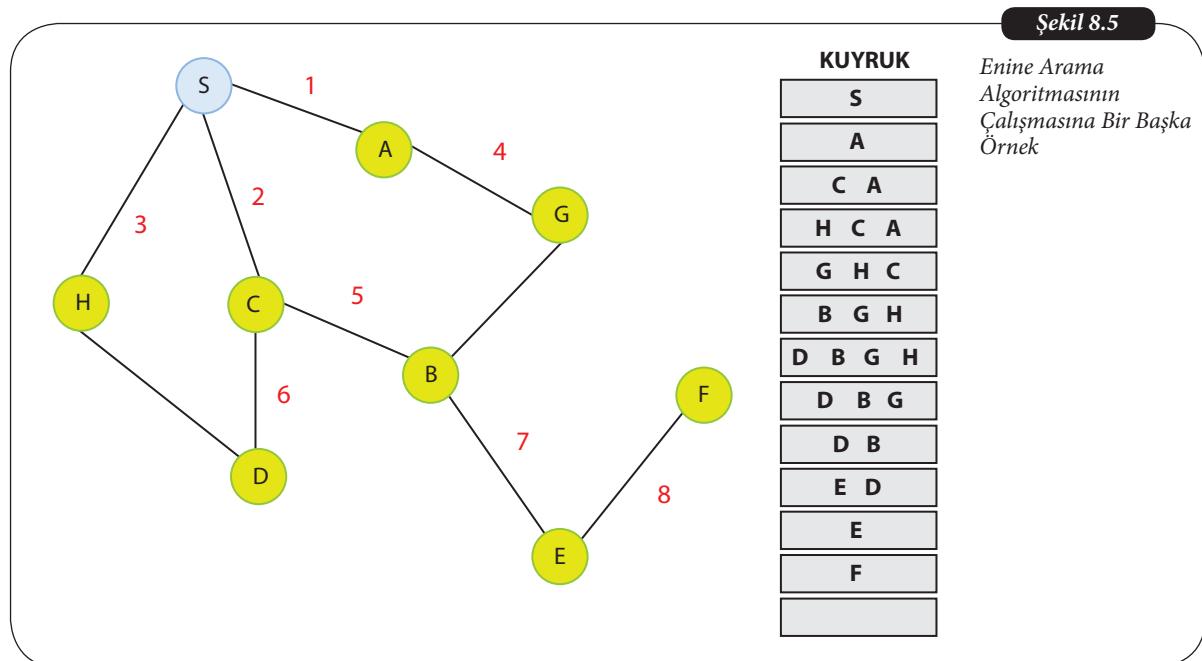
Enine Arama
Algoritmasının
Çalışmasına Bir
Örnek



Enine aramanın çalışma mantığının anlatıldığı bu örnekte, 5 adet düğüm içeren bir çizge bulunmaktadır. Burada, bir düğüm hiç ziyaret edilmemişse sarı renk ile gösterilmektedir ve dolayısıyla algoritma çalışmadan önce bütün düğümler sarı renktedir. Bir düğüm ilk defa ziyaret edildiğinde, mavi renkli olarak gösterilmektedir. Bir düğümün kendisi ve etrafındaki bütün komşuları ziyaret edildiğinde ise o düğüm kırmızı olarak gösterilmektedir. Arama işlemleri yapılrken kuyruk veri yapısından faydalankmaktadır. Şekil 8.4'te kuyruk veri yapısının anlık içeriğine yer verilmiştir. Algoritma ilk çalıştığında, aramaya çizgenin S isimli düğümünden başlanmıştır. İlk olarak S düğümü ziyaret edildiği için mavi olarak gösterilmektedir. Sonraki adımda S düğümünün komşuları sırayla ziyaret edilir. İkinci adımda A düğümü ziyaret edilerek mavi hale getirilir ve bu ziyaretin bilgisine de kuyruk veri yapısı içerisinde yer verilir. Üçüncü ve dördüncü adımlarda S düğümünün

diğer iki komşusu olan B ve C düğümleri ziyaret edilir. Şekilden de görüleceği üzere, artık S düğümünün komşuları içinde ziyaret edilmemiş başka bir düğüm bulunmamaktadır. Bu sebeple S düğümü kırmızı hale gelir ve kuyruk veri yapısı içerisinde bir eleman çıkartılır. Bu eleman, kuyruğun en başındaki A düğümüdür. Daha sonra A düğümünün komşusu olan D düğümü ziyaret edilir ve ziyaret edilmeyen başka komşusu olmadığı için A düğümü kırmızı hale gelir. Algoritma B ve C düğümlerinin komşularının ziyaret edilmesiyle devam edilecektir. Ancak, D düğümü zaten ziyaret edilmiş durumda olduğundan bu iki düğüm de kırmızı hale gelir. Son olarak, D üzerinde işlem yapıldığında algoritma sonlanır. Algoritma sona erdiğinde, kuyruk veri yapısının içinde hiçbir düğüm bulunmayacaktır. Şekilden de görüleceği gibi, 5 düğümün tamamı erişilebilir durumda olduğundan hepsi ziyaret edilmesi mümkün olmuştur. Enine arama algoritması, bu örnekte sırasıyla S , A , B , C , D düğümlerini ziyaret etmektedir.

Şekil 8.5'te, 9 düğümden oluşan bir çizge üzerinde enine arama algoritmasının çalışması kısaca gösterilmiştir. Bu örnekte de bir düğümün birden fazla komşusu olduğunda, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir.



Söz konusu örnekte, şeklin sağ tarafında enine arama çalışırken kuyruk veri yapısının içeriğindeki değişiklik görülmektedir. Şeklin üzerindeki kırmızı sayılar ise algoritmanın çalışması esnasında düğümlerin ziyaret edilme sırasını belirtmektedir. Enine arama algoritması, şekilden görüleceği gibi sırasıyla S , A , C , H , G , B , D , E , F düğümlerini ziyaret etmektedir. Şekil 8.4'teki çizge üzerinde enine arama örneğinin C program kodu ile ifade edilişi Örnek 8.1'de yer almaktadır.

ÖRNEK 8.1*Çizge üzerinde enine aramaya yönelik örnek kod*

```
/* enine_arama.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAKSIMUM 5

struct Dugum
{
    char isim;
    bool ziyaretedildimi;
};

// kuyruk yapisiyla ilgili degiskenler
int kuyruk[MAKSIMUM];
int arkasi = -1;
int onu = 0;
int kuyrukelemanisayisi = 0;

// cizge degiskenleri

// dugumler dizisi
struct Dugum* dugumlistesi[MAKSIMUM];
// komsuluk matrisi
int komsulukmatrisi[MAKSIMUM][MAKSIMUM];
// dugum noktasi sayisi
int dugumsayisi = 0;

//kuyruk fonksiyonlari
void kuyruga_ekle(int veri)
{
    kuyruk[++arkasi] = veri;
    kuyrukelemanisayisi++;
}

int kuyruktan_cikar()
{
    kuyrukelemanisayisi--;
    return kuyruk[onu++];
}

bool kuyruk_bos()
{
    return kuyrukelemanisayisi == 0;
}

// cizge ile ilgili fonksiyonlar

// dugum ekle
void dugum_ekle(char isim)
{
    struct Dugum* dugum = (struct Dugum*) malloc(sizeof(struct Dugum));
    dugum->isim = isim;
    dugum->ziyaretedildimi = false;
    dugumlistesi[dugumsayisi++] = dugum;
}
```

```
// yeni çift yonlu kenar baglantisi ekle
void yonsuz_kenarbaglantisi_ekle(int baslangic, int bitis)
{
    komsulukmatrisi[baslangic][bitis] = 1;
    komsulukmatrisi[bitis][baslangic] = 1;
}

// yeni tek yonlu kenar baglantisi ekle
void yonlu_kenarbaglantisi_ekle(int baslangic, int bitis)
{
    komsulukmatrisi[baslangic][bitis] = 1;
}

// dugum goruntule
void dugum_goruntule(int dugumindeksi)
{
    printf("%c ", dugumlistesi[dugumindeksi]->isim);
}

// ziyaret edilmeyen komsu dugumu getir
int ziyaretedilmeyen_komsuyu_getir(int dugumindeksi)
{
    int i;
    for (i = 0; i<dugumsayisi; i++)
    {
        if (komsulukmatrisi[dugumindeksi][i] == 1 &&
dugumlistesi[i]->ziharetedildimi == false)
            return i;
    }
    return -1;
}

void enine_arama_algoritmasi()
{
    int i;

    // baslangic dugumunun durumunu ziyaret edildi olarak işaretle
    dugumlistesi[0]->ziharetedildimi = true;

    //dugum goruntule
    dugum_goruntule(0);

    // dugum indeksini kuyruga ekle
    kuyruga_ekle(0);
    // ziyaret edilmemis gudum
    int ziyaretedilmemisdugum;

    while (!kuyruk_bos())
    {
        // islem yapilacak ziyaret edilmemis dugumu getir
        int islemyapilandugum = kuyruktan_cikar();

        // islem yapilan dugumun tum komsu dugumlerini ziyaret et
        while ((ziyaretedilmemisdugum =
ziyaretedilmeyen_komsuyu_getir(islemyapilandugum)) != -1)
        {
            dugumlistesi[ziyaretedilmemisdugum]-
>ziharetedildimi = true;
            dugum_goruntule(ziyaretedilmemisdugum);
            kuyruga_ekle(ziyaretedilmemisdugum);
        }
    }
}
```

```

    // kuyruk bos, arama tamamlandi, degiskenen degerini ilk
    haline getir
    for (i = 0; i<dugumsayisi; i++)
    {
        dugumlistesi[i]->ziyaretedildimi = false;
    }
}

void main()
{
    int i, j;
    // komsuluk matrisini baslangicta sifirla
    for (i = 0; i<MAKSIMUM; i++)
    {
        for (j = 0; j<MAKSIMUM; j++)
            komsulukmatrisi[i][j] = 0;
    }
    // dugumleri ekle
    dugum_ekle('S'); // 0
    dugum_ekle('A'); // 1
    dugum_ekle('B'); // 2
    dugum_ekle('C'); // 3
    dugum_ekle('D'); // 4
    // kenar baglantilarini ekle
    yonsuz_kenarbaglantisi_ekle(0, 1); // S - A
    yonsuz_kenarbaglantisi_ekle(0, 2); // S - B
    yonsuz_kenarbaglantisi_ekle(0, 3); // S - C
    yonsuz_kenarbaglantisi_ekle(1, 4); // A - D
    yonsuz_kenarbaglantisi_ekle(2, 4); // B - D
    yonsuz_kenarbaglantisi_ekle(3, 4); // C - D

    printf("\nEnine Arama Sonucları: ");
    enine_arama_algoritması();
    getch();
}

```

Enine arama algoritmasının çalışma mantığının anlatıldığı Örnek 8.1'de çizge ve kuyruk veri yapısı için kullanılan değişken ve fonksiyonlar bulunmaktadır. Öncelikle, kuyruk veri yapısı kuyruk isimli bir değişken kullanılarak gerçekleştirilmektedir. Şekil 8.4 ve 8.5'te görülen kuyruğa ekleme, çıkarma işlemleri için `kuyruga_ekle` ve `kuyruktan_cikar` gibi fonksiyonlar bulunmaktadır. Bir diğer fonksiyon olan `kuyruk_bos` ise kuyruğun içerisindeki eleman kalıp kalmadığını kontrol etmek için kullanılmaktadır. Kodların devamında ise çizge için kullanılan fonksiyonlar bulunmaktadır. Örneğin, ilk başta çizgenin oluşturulması sırasında `dugum_ekle` ve `yonsuz_kenarbaglantisi_ekle` fonksiyonları kullanılmaktadır. Kodlarda enine arama algoritmasının çalışması sırasında kullanılmak üzere `dugum_goruntule` ve `ziyaretedilmeyen_komsuyu_getir` fonksiyonları yer almaktadır. Enine arama algoritması, program kodu içerisinde `enine_arama_algoritması` fonksiyonu sayesinde gerçekleştirilmektedir. Bu fonksiyon, çalışmaya en baştaki düğümün ziyaret edilmesi ve kuyruk veri yapısı içerisinde `kuyruga_ekle` fonksiyonu ile eklenmesiyle başlamaktadır. Daha sonra, `while` döngüsü içerisindeki kod bloğu çalışmaka ve kuyruktaki eleman bulunduğu sürece, `kuyruktan_cikar` fonksiyonu vasıtıyla üzerinde işlem yapılacak düğüm kuyruktan çıkarılarak belirlenmektedir. Üzerinde işlem yapılan düğümün tüm ziyaret edilmemiş komşuları `ziyaretedilmeyen_komsuyu_getir` fonksiyonu yardımıyla tespit edilmekte ve kuyruğa eklenerek "ziyaret edildi" olarak işaretlenmektedir. Bu düğümler aynı zamanda `dugum_goruntule` fonksiyonu yardımıyla ekrana yazdırılmaktadır. Kuyruk veri yapısı içerisindeki tüm düğümler çıkarıldığında, `kuyruk_bos()` fonksiyonunun yardımıyla `while` döngüsü sonlanır ve enine arama algoritmasının çalışması tamamlanır.

`enine_arama.c` program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyi verdigini kontrol ediniz.



SIRA SİZDE

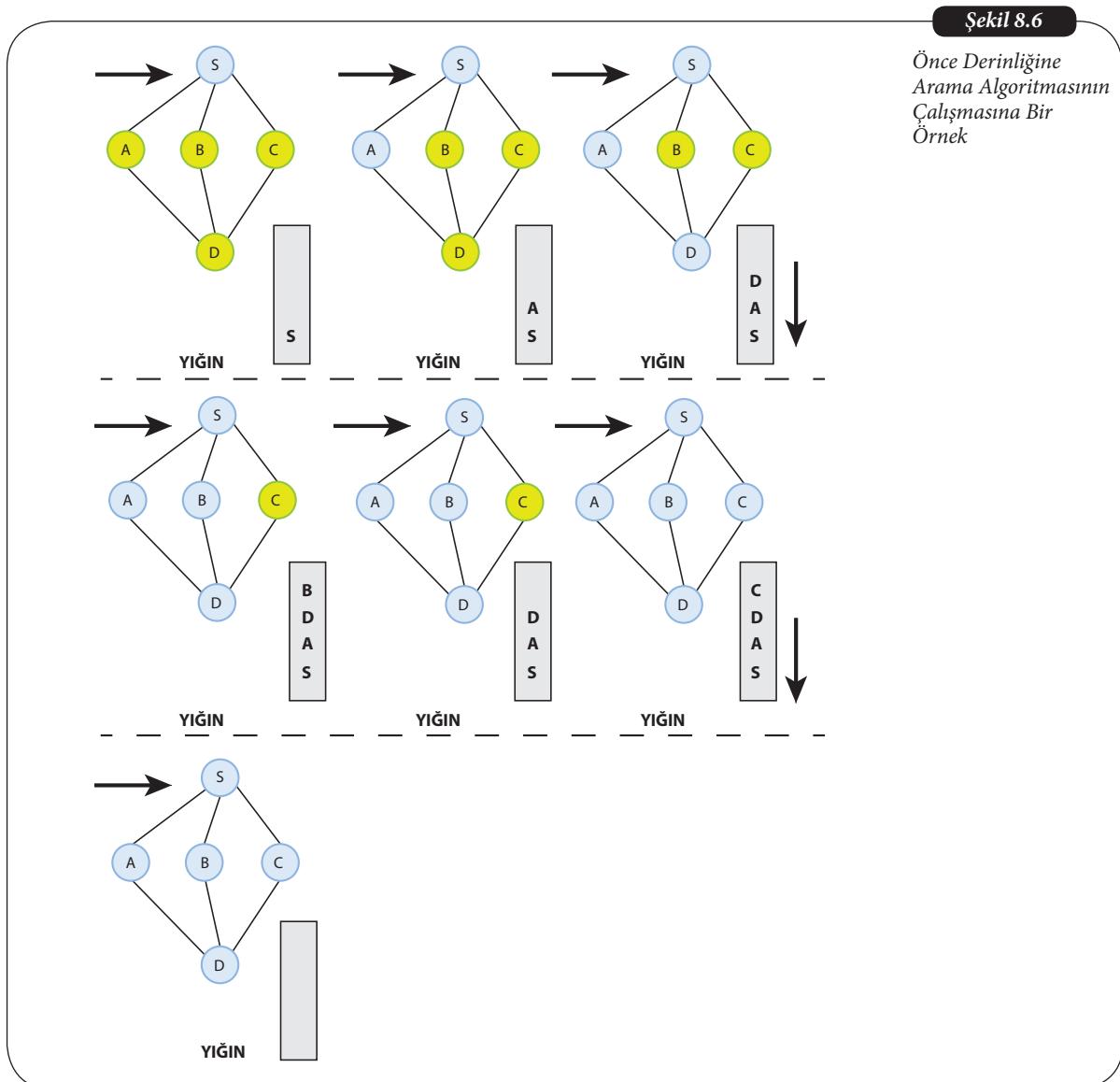
- MAKSİMUM isimli değişkenin boyutunu Şekil 8.5'teki toplam düğüm sayısını göz önüne alarak güncelleyiniz.
- main fonksiyonunun içeriğini Şekil 8.5'teki örnek çizge için çalışacak şekilde düzenleyiniz ve enine arama sırasında çizgenin düğümlerinin ziyaret sıralamasının ekranda nasıl sonuç verdiği gözleyiniz.

ÖNCE DERİNLİĞİNE ARAMA ALGORİTMASI

Önce derinliğine arama, çizgenin bir düğümünden başlanarak bu düğümün komşusu üzerinden gidileBILECEK en uzak düzgüme kadar olan noktaların ziyaret edildiği ve daha sonra geri döñülerek aynı işlemlerin ziyaret edilmemiş düğümler için sürdürdügü bir arama algoritmasıdır. Algoritmanın uygulanması esnasında yiğin (stack) veri yapısından faydalanyılır. Şekil 8.6'da enine arama algoritmasının çalışmasına bir örnek verilmiştir.

Şekil 8.6

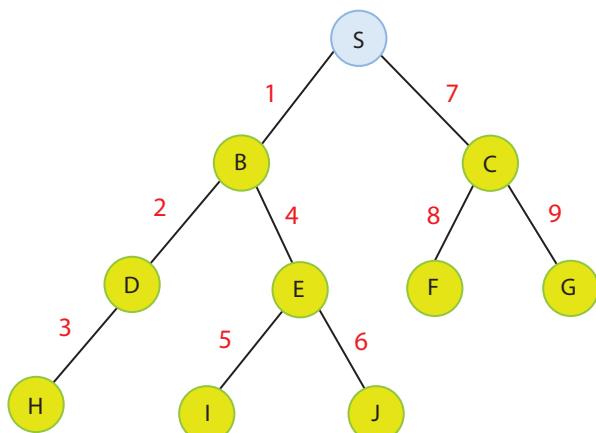
Önce Derinliğine Arama Algoritmasının Çalışmasına Bir Örnek



Önce derinliğine aramanın çalışma mantığının anlatıldığı Şekil 8.6'daki örnekte, 5 adet düğüm içeren bir çizge bulunmaktadır. Şekilde, bir düğüm hiç ziyaret edilmemişse sarı renk ile gösterilmektedir ve dolayısıyla algoritma çalışmadan önce bütün düğümler sarı durumdadır. Bir düğüm, ziyaret edildiğinde mavi olarak gösterilmeye başlanmaktadır. Örnekte, algoritmanın akışı sırasında komşular arasında eşit öncelikli düğümler varsa, aramaya alfabetik sıraya göre devam edilmektedir. Örneğin, A veya B düğümünden devam edilmesi mümkün ise öncelikli olarak A düğümünden devam edilecektir. Arama işlemi yapılrken yiğin veri yapısından faydalananmaka olup, yiğin veri yapısının anlık içeriğine şekilde yer verilmiştir. Algoritma ilk çalıştığında aramaya çizgenin S isimli düğümünden başlanmıştır ve S düğümü yiğinin içerisinde eklenerken mavi olarak gösterilmişdir. Daha sonra, komşular arasında alfabetik olarak en onde gelen A düğümü ile devam edilmiştir. Bu düğüm de yine yiğine eklendikten sonra A'nın komşusu olan D düğümüyle devam edilmiştir. Bu aşamada, D düğümü de yiğine eklenmiş ve şekilde mavi hale getirilmiştir. D düğümünün komşuları B ve C düğümleridir. Alfabetik sıra göz önüne alınlığında B düğümü ile ziyarete devam edilmiştir ve B düğümü de yiğine eklenmiştir. Bu aşamadan sonra, B düğümünün ziyaret edilmemiş komşusu bulunmadığı için bu düğüm yiğinden çıkarılmış ve bir önceki düşüme geri dönülmüştür. Bir önceki düğüm olan D düğümünün ziyaret edilmemiş komşusu durumundaki C düğümü ziyaret edilmiş ve yiğine eklenmiştir. Bu işlemlerin sonunda tüm düğümler ziyaret edilmiş durumdadır. Sonrasında, yiğindaki elemanlar tek tek çıkarılmakta ve çıkarılanın ilgili düğümün ziyaret edilmemiş komşusu olup olmadığına bakılmaktadır. Düğümlerin tamamı ziyaret edilmiş durumda olduğundan, yiğindaki tüm elemanlar sırayla çıkarılmıştır. Son olarak, yiğindeki kalmadığı ve aramanın sona erdiği görülmektedir. Önce derinliğine arama algoritması, bu örnekte sırasıyla S, A, D, B, C düğümlerini ziyaret etmektedir. 10 düğümden oluşan bir çizge üzerinde önce derinliğine arama algoritmasının çalışması Şekil 8.7'de kısaca gösterilmiştir.

Şekil 8.7

Once Derinliğine
Arama Algoritmasının
Çalışmasına Bir Başka
Örnek



Şeklin üzerindeki kırmızı sayılar, algoritmanın çalışmasında düğümlerin ziyaret edilme sırasını belirtmektedir. Önce derinliğine arama algoritması, bu örnekte sırasıyla S, B, D, H, E, I, J, C, F, G düğümlerini ziyaret etmektedir. Şekil 8.6'daki çizge üzerinde önce derinliğine arama örneğinin C program kodu ile ifade edilişi Örnek 8.2'de yer almaktadır.

Çizge üzerinde önce derinliğine aramaya yönelik örnek kod

ÖRNEK 8.2

```
/* once_derinligine_arama.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAKSIMUM 5

struct Dugum
{
    char isim;
    bool ziyaretedildimi;
};

// yigin yapisiyla ilgili degiskenler
int yigin[MAKSIMUM];
int ust = -1;

// cizge degiskenleri

// dugumler dizisi
struct Dugum* dugumlistesi[MAKSIMUM];
// komsuluk matrisi
int komsulukmatrisi[MAKSIMUM][MAKSIMUM];
// dugum noktasi sayisi
int dugumsayisi = 0;

// yigin fonksiyonlari
void yigina_ekle(int veri)
{
    yigin[++ust] = veri;
}
int yigindan_cikar()
{
    return yigin[ust--];
}
int yigin_enusstekini_goster()
{
    return yigin[ust];
}
bool yigin_bos()
{
    return ust == -1;
}

// cizge ile ilgili fonksiyonlar
// dugum ekle
void dugum_ekle(char isim)
{
    struct Dugum* dugum = (struct Dugum*) malloc(sizeof(struct Dugum));
    dugum->isim = isim;
    dugum->ziyaretedildimi = false;
    dugumlistesi[dugumsayisi++] = dugum;
}

// yeni cift yonlu kenar baglantisi ekle
void yonsuz_kenarbaglantisi_ekle(int baslangic, int bitis)
{
    komsulukmatrisi[baslangic][bitis] = 1;
    komsulukmatrisi[bitis][baslangic] = 1;
}
```

```

// yeni tek yonlu kenar baglantisi ekle
void yonlu_kenarbaglantisi_ekle(int baslangic, int bitis)
{
    komsulukmatrisi[baslangic][bitis] = 1;
}

// dugum goruntule
void dugum_goruntule(int vertexIndex)
{
    printf("%c ", dugumlistesi[vertexIndex]->isim);
}

// ziyaret edilmeyen komsu dugumu getir
int ziyaretedilmeyen_komsuyu_getir(int dugumindeksi)
{
    int i;
    for (i = 0; i<dugumsayisi; i++)
    {
        if (komsulukmatrisi[dugumindeksi][i] == 1 &&
dugumlistesi[i]->ziyaretedildimi == false)
            return i;
    }
    return -1;
}

void once_derinligine_arama_algoritmasi()
{
    int i;
    // baslangic dugumunun durumunu ziyaret edildi olarak isaretle
    dugumlistesi[0]->ziyaretedildimi = true;

    //dugum goruntule
    dugum_goruntule(0);

    // dugum indeksini kuyruga ekle
    yigina_ekle(0);
    // ziyaret edilmemis dugum
    int ziyaretedilmemisdugum;
    while (!yigin_bos())
    {
        // ziyaret edilmemis kuyrugu getir
        ziyaretedilmemisdugum = ziyaretedilmeyen_komsuyu_
getir(yigin_enusstekini_goster());

        if (ziyaretedilmemisdugum == -1)
        {
            yigindan_cikar();
        }
        else
        {
            dugumlistesi[ziyaretedilmemisdugum]-
>ziyaretedildimi = true;
            dugum_goruntule(ziyaretedilmemisdugum);
            yigina_ekle(ziyaretedilmemisdugum);
        }
    }
    // yigin bos, arama tamamlandi, degiskenin degerini ilk haline
    getir
    for (i = 0; i<dugumsayisi; i++)
    {
        dugumlistesi[i]->ziyaretedildimi = false;
    }
}

```

```

void main()
{
    int i, j;
    // komsuluk matrisini baslangicta sifirla
    for (i = 0; i<MAKSIMUM; i++)
    {
        for (j = 0; j<MAKSIMUM; j++)
            komsulukmatrisi[i][j] = 0;
    }

    dugum_ekle('S');    // 0
    dugum_ekle('A');    // 1
    dugum_ekle('B');    // 2
    dugum_ekle('C');    // 3
    dugum_ekle('D');    // 4

    yonsuz_kenarbaglantisi_ekle(0, 1);    // S - A
    yonsuz_kenarbaglantisi_ekle(0, 2);    // S - B
    yonsuz_kenarbaglantisi_ekle(0, 3);    // S - C
    yonsuz_kenarbaglantisi_ekle(1, 4);    // A - D
    yonsuz_kenarbaglantisi_ekle(2, 4);    // B - D
    yonsuz_kenarbaglantisi_ekle(3, 4);    // C - D

    printf("\nOnce Derinligine Arama Sonucları: ");
    once_derinligine_arama_algoritması();
    getch();
}

```

Önce derinliğine arama algoritmasının çalışma mantığının anlatıldığı Örnek 8.2'de, çizge ve yiğin veri yapısı için kullanılan değişken ve fonksiyonlar bulunmaktadır. Önce-likle, yiğin veri yapısı `yigin` isimli bir değişken kullanılarak gerçekleştirilmektedir. Şekil 8.6'da görülen yiğina ekleme, çıkarma işlemleri için `yigina_ekle` ve `yigindan_cikar` gibi fonksiyonlar bulunmaktadır. Bir diğer fonksiyon olan `yigin_bos` ise yiğinin içerisinde eleman kalıp kalmadığını kontrol etmek için kullanılmaktadır. Daha sonrasında ise çizge için kullanılan fonksiyonlar bulunmaktadır. Örneğin, ilk başta çizgenin oluşturulması sırasında `dugum_ekle` ve `yonsuz_kenarbaglantisi_ekle` fonksiyonları kullanılmaktadır. Önce derinliğine arama algoritmasının çalışması sırasında kullanılmak üzere `dugum_goruntule` ve `ziyaretedilmeyen_komsuyu_getir` fonksiyonları yer almaktadır. Önce derinliğine arama algoritması, program kodlarındaki `once_derinligine_arama_algoritması` fonksiyonu sayesinde gerçekleştirilmektedir. Bu fonksiyon, çalışmaya en baştaki düğümün ziyaret edilmesi ve yiğin veri yapısı içerisine `yigina_ekle` fonksiyonu ile eklenmesiyle başlamaktadır. Daha sonra, `while` döngüsü içerisindeki kod bloğu çalışmaka ve kuyrukta eleman bulunduğu sürece `yigindan_cikar` fonksiyonu vasıtasiyla üzerinde işlem yapılacak düğüm, yiğinden çıkararak belirlenmektedir. Özellikle üzerinde işlem yapılan düğümden başlanarak, belli bir yönde gidilebilen en uzak derinliğe kadar olan tüm ziyaret edilmemiş komşular ziyaret edilmektedir. Ziyaret edilmiş bu düğümler aynı zamanda `dugum_goruntule` fonksiyonu yardımıyla ekrana yazdırılmaktadır. Program koduna göre, `ziyaretedilmeyen_komsuyu_getir` fonksiyonunun döndürdüğü değer -1 olmadığı sürece ziyaret edilmemiş komşular bulunmaktadır. Bu fonksiyon -1 değeri döndürdüğünde ise artık yiğinden eleman çıkararak üst düğümle-re doğru geri dönülmekte ve bu kez onların komşuları arasında ziyaret edilmemiş düğüm olup olmadığı incelenmektedir. Yiğin veri yapısı içerisindeki tüm düğümler çıkarıldığında `yigin_bos()` fonksiyonunun yardımıyla `while` döngüsü sonlanmakta ve önce derinliğine arama algoritmasının çalışması tamamlanmaktadır.

SIRA SİZDE

2

once_derinligine_arama.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyi verdigini kontrol ediniz.

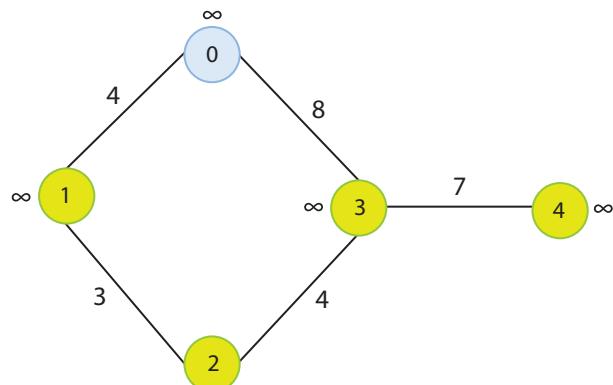
- MAKSIMUM isimli değişkenin boyutunu Şekil 8.7'deki toplam düğüm sayısını göz önüne alarak güncelleyiniz.
- main fonksiyonunun içeriğini Şekil 8.7'deki örnek çizge için çalışacak şekilde düzenleyiniz ve önce derinliğine arama sırasında çizgenin düğümlerinin ziyaret sıralamasının ekranda nasıl sonuç verdiği gözleyiniz.

DİJKSTRA EN KISA YOL ALGORİTMASI

Dijkstra algoritması, ağırlıklandırılmış çizgelerde bir başlangıç düğümü ile diğer düğümler arasındaki en kısa mesafeyi tespit etmek için kullanılır. Bu algoritmanın amacını açıklamak için örnek olarak Türkiye haritasındaki şehirlerin bir çizge üzerinde düğümler olarak gösterildiğini varsayıyalım. Düğümler arasındaki ağırlıklandırılmış kenar bağlantıları ise her iki şehir çifti arasındaki uzaklıği temsil edecektir. Bu bilgilerin yardımıyla Dijkstra algoritması kullanılarak, belirtilen bir şehir ile diğer bütün şehirler arasındaki en kısa yollar bulunabilir. Sonuçta, çıktı olarak farklı şehirler arasındaki en kısa mesafeler ve izlenmesi gereken yollar elde edilecektir. Bu algoritmanın başlangıcında, başlangıç düğümü ile diğer düğümler arasındaki uzaklıkların tamamının sonsuz olduğu varsayılar. Algoritma, düğümler arasındaki en kısa yolları aradığı için düğümler arasında daha kısa yollar bulunukça, sonsuz değeri ilgili yolu uzunluğu ile değiştirilir. Şekil 8.8'de Dijkstra algoritması ile en kısa yolu bulunancağı bir çizge örneği verilmiştir. Bu örnekte, 0 düğümü ile diğer düğümler arasındaki en kısa yol aranacağı için 0 düğümü mavi ile gösterilmektedir.

Şekil 8.8

Dijkstra
Algoritmasının
Çalışmasına Bir
Örnek



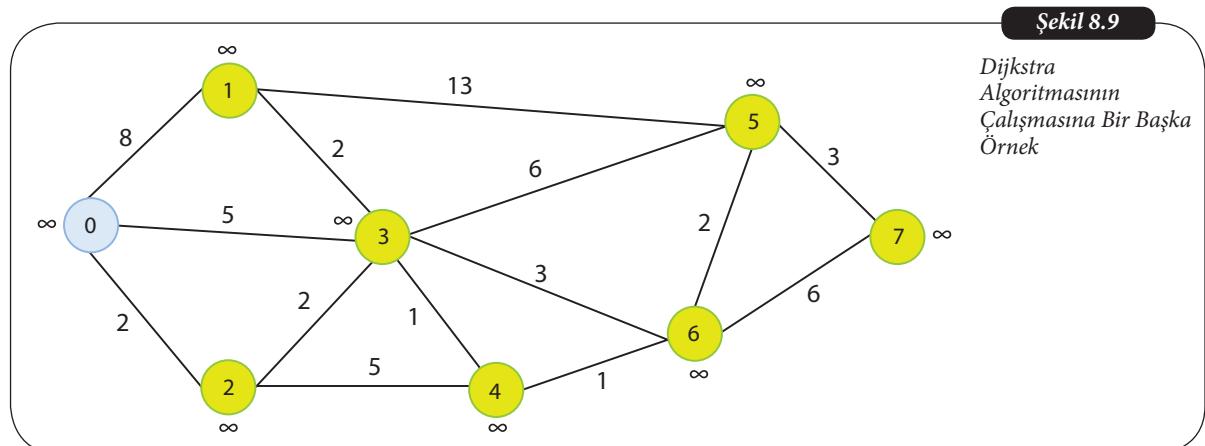
Dijkstra algoritması, her bir adımda ziyaret edilenler listesine yeni bir düğüm eklemekte ve düğümler arası mesafeleri güncellemektedir. Tablo 8.2'de algoritmanın adım adım ilerleyisi ve 0 düğümü ile diğerleri arasındaki mesafeler belirtilmektedir. Mesafe değeri kesinleşen ve üzerinde artık işlem yapılmayacak düğümleri gösteren hücreler, tablo içerisinde mavi olarak görüntülenmektedir. Ayrıca, parantez içerisinde belirtilen değerler, o mesafenin elde edilmesi için ziyaret edilmesi gerekliliği olan düğümü göstermektedir. Tabloda ilk olarak 0 düğümü ziyaret edilmiştir. 0 düğümü ile diğer düğümlerin arasındaki mesafeler incelenmiş ve ilk satırın değerleri oluşturulmuştur. 0 düğümü doğrudan sadece 1 ve 3 düğümleri ile komşu olduğu için bu tablo hücrelerine 4 ve 8 değerleri verilmiştir. 2 ve 4 düğümleri ile komşu olmadığı için onlar ile arasındaki mesafe sonsuzdur. İkinci işlem adımına başlarken, ilk satırındaki mavi olmayan değerler içerisinde en küçük olan tespit

edilir. 4, 8 ve sonsuz değerleri içerisinde en küçükü 4 olduğu için bir sonraki adımda 1 düğümünün ziyaret edilmesi söz konusudur. Çünkü 4 değeri 1 düğümüne ait sütunda yer almaktadır. 1 düğümünü ziyaret ederken ilgili satırdaki değerlerin hangilerinin güncelleneceği tespit edilmelidir. 1 düğümü, 2 ve 0 düğümleri ile komşu durumdadır. Bu sebeple, 0 ile 2 düğümleri arasındaki mesafenin 4 ve 3'ün toplamı olan 7 değeri olduğu tespit edilmiş ve tabloya yazılmıştır. Daha sonra, ikinci satırdaki mavi olmayan değerler içerisinde en küçük olan tespit edilir. 7, 8 ve sonsuz değerleri arasından en küçükü 7 olduğu için bir sonraki adımda 2 düğümünün ziyaret edilmesi söz konusudur. 2 düğümü, 1 ve 3 düğümü ile komşu durumdadır. 0, 1 ve 2 düğümlerini kullanarak 3'e ulaşmak mümkündür. Ancak, şekilde de görüleceği üzere ilgili mesafe 4, 3 ve 4 sayılarının toplamı olan 11 sayısına eşittir. Bu değer 8'den büyük olduğu için 3 düğümüne ulaşmak için gerekli en küçük mesafe bir değişiklik yapılmayacaktır. Böyle bir durumda, mesafe değeri olarak daha küçük bir sayı elde edilebilseydi tablonun güncellenmesi gereklidir. Daha sonra, mevcut satırda mavi olmayan 8 ve sonsuz arasından en küçük olan değerin tespit edilecektir. 3 düğümü ile 4 düğümü arasında komşuluk bulunduğu için tabloya 15 sayısı yazılmıştır. Bunun sebebi, 3 düğümüne ulaşmak için en kısa mesafenin 8 olduğunu bilinmesi ve bu sayının son iki düğüm arasındaki mesafe olan 7 sayısı ile toplanmış olmasıdır. Yani 0 ve 3 düğümleri ziyaret edilerek 4 düğümüne ulaşılmıştır. Algoritmanın çalışması tamamlanmıştır ve son satırdaki değerler 0 düğümünden her bir sütunda bulunan ilgili düğümlere ulaşmak için en kısa yollardır. Örneğin, 0 ile 2 düğümü arasındaki en kısa mesafe 7'dir. Bunun için öncesinde 1 düğümünün ziyaret edilmesi gereklidir.

Ziyaret sırası	0	1	2	3	4
0	0(0)	4(0)	$\infty(0)$	8(0)	$\infty(0)$
1	0(0)	4(0)	7(1)	8(0)	$\infty(0)$
2	0(0)	4(0)	7(1)	8(0)	$\infty(0)$
3	0(0)	4(0)	7(1)	8(0)	15(3)
4	0(0)	4(0)	7(1)	8(0)	15(3)

Tablo 8.2
Dijkstra Algoritmasının Çalışması Sırasındaki Mesafe Bilgileri

Şekil 8.9'de Dijkstra algoritması ile en kısa yolu bulunacağı bir başka çizge örneği verilmiştir. Bu örnekte, 0 düğümü ile diğer düğümler arasındaki en kısa yol aranacağı için 0 düğümü mavi ile gösterilmektedir.



Dijkstra algoritması, her bir adımda ziyaret edilenler listesine yeni bir düğüm eklemekte ve düğümler arası mesafeleri güncellemektedir. Tablo 8.3'te algoritmanın adım adım ilerleyişi ve 0 düğümü ile diğerleri arasındaki mesafeler belirtilmektedir. Mevcut durumda ziyaret edilmiş olan ve tekrar ziyaret edilmeyecek olan düğümleri gösteren hücreler mavi olarak görüntülenmektedir. Ayrıca parantez içerisinde belirtilen değerler o mesafenin elde edilmesi için ziyaret edilmesi gereklili olan düğümü göstermektedir. Tabloda ilk olarak 0 düğümü ziyaret edilmiştir. 0 düğümü ile diğer düğümlerin arasındaki mesafeler incelenmiş ve ilk satırın değerleri oluşturulmuştur. 0 düğümü, doğrudan sadece 1, 2 ve 3 düğümleri ile komşu olduğu için bu tablo hücrelerine 8, 2 ve 5 değerleri verilmiştir. 0 düğümünden diğer düğümlere doğrudan erişim olmadığı için ilk satırdaki diğer hücrelere sonsuz değeri atanmıştır. 0 düğümü ile 0 düğümü arasında bir mesafe söz konusu olmadığı için ilk sütun değeri 0'dır. İkinci işlem adımına başlarken, ilk satırdaki mavi olmayan değerler arasından en küçük olanı tespit edilecektir. 8, 2, 5 ve sonsuz değerleri arasından en küçüğü 2 olduğu için bir sonraki adımda 2 düğümünün ziyaret edilmesi söz konusudur. Çünkü 2 değeri, 2 düğümüne ait sütunda yer almaktadır. 2 düğümünü ziyaret ederken ilgili satırdaki değerlerin hangilerinin güncelleneceği tespit edilmelidir. 2 düğümü, 3 ve 4 düğümleri ile komşu durumdadır. Bu sebeple, 0 ve 3 düğümleri arasındaki mesafenin 2 ile 2'nin toplamı olan 4 olduğu tespit edilmiş ve tabloya yazılmıştır. 3 sütununun daha önceki değeri bu aşamada güncellenmiştir. Bunun sebebi, bu adımda 0 düğümünden 3 düğümüne erişim için öncekinden daha kısa bir yol tespit edilmiş olmasıdır. Aynı durum 4 sütunu için de geçerlidir. 0 ile 4 düğümleri arasında 2 düğümünün kullanılmasıyla daha kısa bir yol mevcuttur. 0, 2 ve 3 düğümleri ziyaret edilerek 4 düğümüne ulaşılabilir. Bu adımda iki düğüm arasındaki mesafe, 7 yerine 5 olarak güncellenmiştir. Bu ziyaretlere, ziyaret edilmemiş düğümler içerisinde en kısa mesafeye sahip olanlar ile devam edilir ve tablonun son satırındaki değerler elde edilir. Bu aşamalarda, ilgili sütunlar için daha kısa yollar bulunmuşsa tablodaki o sütunlara ait değerler güncellenir.

Tablo 8.3
Dijkstra Algoritması Çalışması Sırasındaki Mesafe Bilgileri

Ziyaret sırası	0	1	2	3	4	5	6	7
0	0 (0)	8 (0)	2 (0)	5 (0)	∞ (0)	∞ (0)	∞ (0)	∞ (0)
2	0 (0)	8 (0)	2 (0)	4 (2)	7 (2)	∞ (0)	∞ (0)	∞ (0)
3	0 (0)	6 (3)	2 (0)	4 (2)	5 (3)	10 (3)	7 (3)	∞ (0)
4	0 (0)	6 (3)	2 (0)	4 (2)	5 (3)	10 (3)	6 (4)	∞ (0)
1	0 (0)	6 (3)	2 (0)	4 (2)	5 (3)	10 (3)	6 (4)	∞ (0)
6	0 (0)	6 (3)	2 (0)	4 (2)	5 (3)	8 (6)	6 (4)	12 (6)
5	0 (0)	6 (3)	2 (0)	4 (2)	5 (3)	8 (6)	6 (4)	11 (5)
7	0 (0)	6 (3)	2 (0)	4 (2)	5 (3)	8 (6)	6 (4)	11 (5)

Tabloda son satırdaki değerleri şu şekilde yorumlayabiliriz. Örneğin, 0 ile 5 düğümü arasındaki en kısa mesafe 8'dir. Parantez içerisindeki değerler de hangi yollardan geçilmesi gerektiğini göstermektedir. 0 düğümünden 5 düğümüne 2, 3, 4, 6 düğümleri üzerinden ulaşmak gerekecektir. Bu sonuca şu şekilde ulaşılabilir. Başlığı 5 olan sütun için son satır değeri 8(6)'dır. Bu bilgiden, en son 6 düğümüne uğranması gerektiği anlaşılmaktadır. Başlığı 6 olan sütun ise bu düğüm için en kısa mesafenin nasıl bulunabileceğini gösterir. Başlığı 6 olan sütun için son satır değeri 6(4)'tir. Bu bilgiden, bir adım önce 4 düğümüne uğranması gerektiği anlaşılmaktadır. Aynı şekilde, başlığı 4 olan sütun için son satır değeri 5(3)'tir. Bu bilgiden, bir adım önce 3 düğümüne uğranması gerektiği anlaşılmaktadır. Aynı şekilde, başlığı 3 olan sütun için son satır değeri 4(2)'tir. Bu bilgiden, bir adım önce 2 düğümüne uğranması gerektiği anlaşılmaktadır. Bunları bir araya getirdiğimizde ise 0

düğümünden 5 düğümüne en kısa yolu kullanarak gitmek için 2, 3, 4, 6 düğümlerinin ziyaret edilmesi gerektiği görülmektedir. Bu yol üzerindeki mesafeleri topladığımızda ise 2, 2, 1, 1 ve 2 değerlerinin toplamı olan 8 değerini elde ederiz. Şekil 8.8'deki çizge üzerinde Dijkstra algoritması ile en kısa yolu bulma örneğinin C program kodu ile ifade edilişi Örnek 8.3'te yer almaktadır.

Çizge üzerinde Dijkstra algoritmasıyla en kısa yolu bulmaya yönelik örnek kod

ÖRNEK 8.3

```
/* dijkstra_en_kisa_yol.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAKS 5
#define SONSUZ 9999

struct Dugum
{
    char isim;
    bool ziyaretedildimi;
};

// cizge degiskenleri

// dugumler dizisi
struct Dugum* dugumlistesi[MAKS];
// komsuluk matrisi
int komsulukmatrisi[MAKS][MAKS];
// dugum noktası sayisi
int dugumsayisi = 0;

// cizge ile ilgili fonksiyonlar

// dugum ekle
void dugum_ekle(char isim)
{
    struct Dugum* dugum = (struct Dugum*) malloc(sizeof(struct Dugum));
    dugum->isim = isim;
    dugum->ziyaretedildimi = false;
    dugumlistesi[dugumsayisi++] = dugum;
}

// yeni çift yonlu kenar baglantisi ekle
void yonsuz_kenarbaglantisi_ekle(int baslangic, int bitis,
int mesafe)
{
    komsulukmatrisi[baslangic][bitis] = mesafe;
    komsulukmatrisi[bitis][baslangic] = mesafe;
}

// yeni tek yonlu kenar baglantisi ekle
void yonlu_kenarbaglantisi_ekle(int baslangic, int bitis, int
mesafe)
{
    komsulukmatrisi[baslangic][bitis] = mesafe;
}

// dugum goruntule
void dugum_goruntule(int dugumindeksi)
```

```

        printf("%c ", dugumlistesi[dugumindeksi]->isim);
    }

    // en kisa yol algoritma fonksiyonu
    void dijkstra(int baslangicdugumu)
    {
        int mtablosu[MAKS][MAKS], mesafe[MAKS], onceki[MAKS],
enkisayol[MAKS];
        int ziyaretli[MAKS], sayi, enkmesafe, sonraki, i, j, k;

        //onceki[] her dugumun oncesindeki dugumu saklar
        //en kisa yollari tutan mesafe tablosu matrisini olustur
        for (i = 0; i<MAKS; i++)
            for (j = 0; j<MAKS; j++)
                if (komsulukmatrisi[i][j] == 0)
                    mtablosu[i][j] = SONSUZ;
                else
                    mtablosu[i][j] = komsulukmatrisi[i][j];

        for (i = 0; i<MAKS; i++)
        {
            mesafe[i] = mtablosu[baslangicdugumu][i];
            onceki[i] = baslangicdugumu;
            ziyaretli[i] = 0;
        }

        mesafe[baslangicdugumu] = 0;
        ziyaretli[baslangicdugumu] = 1;
        sayi = 1;

        while (sayi<MAKS - 1)
        {
            enkmesafe = SONSUZ;

            //sonraki en kisa mesafeyi verir
            for (i = 0; i<MAKS; i++)
                if (mesafe[i]<enkmesafe && !ziyaretli[i])
                {
                    enkmesafe = mesafe[i];
                    sonraki = i;
                }

            //daha iyi bir yol olup olmadigini kontrol et
            ziyaretli[sonraki] = 1;
            for (i = 0; i<MAKS; i++)
                if (!ziyaretli[i])
                    if (enkmesafe + mtablosu[sonraki][i] <
mesafe[i])
                    {
                        mesafe[i] = enkmesafe +
mtablosu[sonraki][i];
                        onceki[i] = sonraki;
                    }
            sayi++;
        }

        //en kisa mesafenin elde edildigi yolu ekrana bas
        for (i = 0; i < MAKS; i++)
        {
            for (k = 0; k < MAKS; k++)
            {
                enkisayol[k] = -1;
            }
            k = MAKS;
        }
    }
}

```

```

if (i != baslangicdugumu)
{
    j = i;
    do
    {
        k--;
        j = onceki[j];
        enkisayol[k] = j;
        //printf(" %d", j);
    } while (j != baslangicdugumu);
}

printf("\n%d. dugum icin en kisa mesafe=%d", i,
mesafe[i]);
printf("\nYol = ");
for (k = 0; k<MAKS; k++)
{
    if (enkisayol[k] != -1) printf(" %d",
enkisayol[k]);
}
printf(" %d", i);
}
}

void main()
{
    int i, j, baslangic_dugum;
    // komsuluk matrisini baslangicta sifirla
    for (i = 0; i<MAKS; i++)
    {
        for (j = 0; j<MAKS; j++)
            komsulukmatrisi[i][j] = SONSUZ;
    }
    // dugumleri ekle
    dugum_ekle('0'); // 0
    dugum_ekle('1'); // 1
    dugum_ekle('2'); // 2
    dugum_ekle('3'); // 3
    dugum_ekle('4'); // 4

    // kenar baglantilarini ekle
    yonsuz_kenarbaglantisi_ekle(0, 1, 4); // 0 - 1 = 4
    yonsuz_kenarbaglantisi_ekle(0, 3, 8); // 0 - 3 = 8
    yonsuz_kenarbaglantisi_ekle(1, 2, 3); // 1 - 2 = 3
    yonsuz_kenarbaglantisi_ekle(2, 3, 4); // 2 - 3 = 4
    yonsuz_kenarbaglantisi_ekle(3, 4, 7); // 3 - 4 = 7

    // Baslangic dugumunu gir
    printf("\nBaslangic dugumunu sec: ");
    scanf("%d", &baslangic_dugum);

    // Dijkstra algoritmasini calistir
    dijkstra(baslangic_dugum);
    getch();
}
}

```

Dijkstra en kısa yol algoritmasının çalışma mantığının anlatıldığı Örnek 8.3'te çizge için kullanılan değişken ve fonksiyonlar bulunmaktadır. Örneğin, ilk aşamada çizgenin oluşturulması sırasında `dugum_ekle` ve `yonsuz_kenarbaglantisi_ekle` fonksiyonları kullanılmaktadır. `yonsuz_kenarbaglantisi_ekle` fonksiyonu, kenar bağlantısının hangi iki düğüm arasında olacağı ve ağırlığının ne olduğu gibi farklı parametreler almaktadır. Program içerisinde temsili olarak sonsuz ifadesi yerine 9999 değeri içeren

SONSUZ isimli değişken kullanılmıştır. Dijkstra en kısa yol algoritması, dijkstra fonksiyonu sayesinde gerçekleştirilmektedir. Fonksiyon içerisinde Dijkstra en kısa yol algoritması, temel olarak `while (sayi<MAKS - 1)` kod bloğu içerisinde gerçekleştirilmektedir. Bu kod bloğunda örnek şekillerde gösterildiği gibi adım adım işlem yapılmaktadır. Her işlem admında en küçük mesafeyi barındıran düğümler seçilmekte ve ziyaretlere o düğümler ile devam edilmektedir. Kod bloğunda adım adım her düğüme ulaşmak için en kısa mesafe tespit edilmektedir. Daha sonraki program kodlarında ise bu mesafeyi elde etmek için hangi düğümlerin ziyaret edilmesi gerektiği ekrana yazdırılmaktadır. Program çalışığında kullanıcıdan başlangıç düğümünü girmesi istenmektedir. Başlangıç düğümü olarak 0'ın girildiği durumdaki program kodunun çıktısı aşağıdaki gibidir:

```
Baslangic dugumunu sec: 0
0. dugum icin en kisa mesafe=0
Yol = 0
1. dugum icin en kisa mesafe=4
Yol = 0 1
2. dugum icin en kisa mesafe=7
Yol = 0 1 2
3. dugum icin en kisa mesafe=8
Yol = 0 3
4. dugum icin en kisa mesafe=15
Yol = 0 3 4
```

SIRA SİZDE



`dijkstra_en_kisa_yol.c` program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyi verdiğini kontrol ediniz.

- `MAKS` isimli değişkenin boyutunu **Şekil 8.9'daki toplam düğüm sayısını** göz önüne alarak güncelleyiniz.
- `main` fonksiyonunun içeriğini **Şekil 8.9'daki örnek çizge** için çalışacak şekilde düzenleyiniz ve Dijkstra en kısa yol algoritmasının çalışması sonrasında ekranda nasıl sonuç verdiğiğini gözleyiniz.

Özet



Çizge kavramını tanımlamak

Çizge, düğümlerle bu düğümleri birbirine bağlayan kenarlardan oluşan ve ağ görünümünde olan bir tür veri yapısıdır. Çizgeler, düğümler arasındaki kenar bağlantılarının tipine göre yönlü çizge ve yönssüz çizge olmak üzere ikiye ayrılırlar. Yönlü çizgelerin kenar bağlantılarında yönleri temsil eden oklar bulunurken, yönssüz çizgelerde böyle bir durum söz konusu değildir. Çizgelerde komşuluk ve çıkışım olmak üzere iki farklı ilişki türü vardır. Komşuluk, iki düğüm arasında bir bağlantı olmasını ifade eder. Çıkışım ise düğüm ile kenar bağlantısı arasındaki ilişkiyi belirtir. Çizgelere özgü yol kavramı, çizgenin içerisinde bir düğümden başka düğüme ulaşmak için geçilmesi gereken düğümler olarak tanımlanabilir. Çizgelerde komşuluk ilişkilerini temsil etmek için kullanılan en yaygın yöntemlerden birisi komşuluk matrisidir. Bu hususların haricinde, ağırlıklandırılmış çizge olarak adlandırılan ve düğümler arasındaki kenar bağlantıları üzerinde sıfırdan farklı değerlerin yer aldığı bir çizge türü de mevcuttur.



Çizgelerde arama algoritmalarını uygulamak

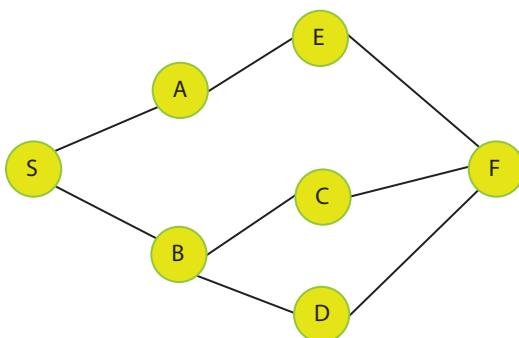
Herhangi bir çizgenin içerdiği düğümler, özel arama algoritmalarıyla bir noktadan başlanarak belirli sırada ziyaret edilebilmekte ve listelenebilmektedir. Çizgeler üzerinde bu işlemin yapılabilmesi için enine arama ve önce derinliğine arama algoritmaları mevcuttur. Enine arama, çizgenin bir düğümünden başlanarak, o düğümün komşu düğümlerinin ve onların da komşularının sırayla ziyaret edildiği arama algoritmasıdır. Bu algoritmanın çalışması sırasında, öncelikle başlangıç düğümünün tüm komşuları ziyaret edilir. Daha sonra, başlangıç düğümünün komşuları ile komşu olan düğümlerden devam edilir. Algoritmanın uygulanması esnasında kuyruk veri yapısından faydalанılır. Önce derinliğine arama, çizgenin bir düğümünden başlanarak bu düğümün komşusu üzerinden gidilebilecek en uzak düzgüme kadar olan noktaların ziyaret edildiği ve daha sonra geri dönülerek aynı işlemlerin ziyaret edilmemiş düğümler için sürdürdüğü bir arama algoritmasıdır. Algoritmanın uygulanması esnasında yığın veri yapısından faydalанılır.



Çizgelerde en kısa yol bulma algoritmalarını kullanmak Dijkstra algoritması, ağırlıklandırılmış çizgelerde başlangıç düğümü ile diğer düğümler arasındaki en kısa mesafeleri tespit etmek için kullanılır. Örneğin, Türkiye haritasındaki şehirlerin bir çizge üzerinde düğümler olarak gösterildiğini varsayılmı. Düğümler arasındaki ağırlıklandırılmış kenar bağlantıları ise her iki şehir çifti arasındaki uzaklıği temsил etmektedir. Bu bilgilerin yardımıyla, Dijkstra algoritması kullanılarak belirtilen bir şehir ile diğer bütün şehirler arasındaki en kısa yollar bulunabilir. Böylelikle, çıktı olarak farklı şehirler arasındaki en kısa mesafeler ve izlenmesi gereken yollar elde edilir. Bu algoritmanın başında, başlangıç düğümü ile diğer düğümler arasındaki uzaklıkların tamamının sonsuz olduğu varsayılr. Algoritma, düğümler arasındaki en kısa yolları aradığı için düğümler arasında daha kısa yollar bulunukça sonsuz ifadesi bu yolu mesafesi ile değiştirilir.

Kendimizi Sınavalım

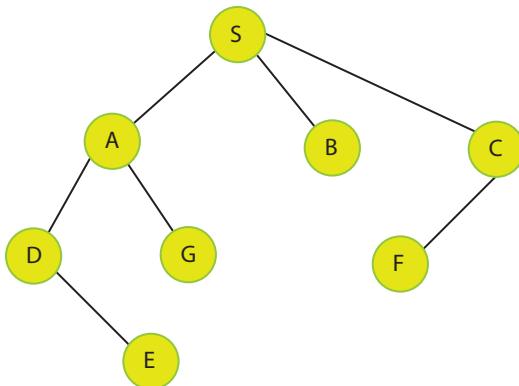
1.



Yukarıdaki çizge üzerinde, başlangıç noktası S olacak şekilde enine arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu varsa, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir. Buna göre çizgedeki düğümlerin ziyaret edilme sırası aşağıdakilerden hangisidir?

- a. S, A, E, B, C, F, D
- b. S, A, E, F, B, C, D
- c. S, A, E, B, C, D, F
- d. S, A, B, C, E, D, F
- e. S, A, B, E, C, D, F

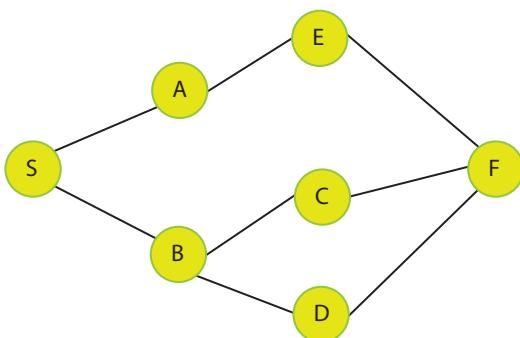
2.



Yukarıdaki çizge üzerinde, başlangıç noktası S olacak şekilde enine arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu varsa, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir. Buna göre çizgedeki düğümlerin ziyaret edilme sırası aşağıdakilerden hangisidir?

- a. S, A, B, C, D, G, F, E
- b. S, A, B, C, D, G, E, F
- c. S, A, B, C, D, F, G, E
- d. S, A, B, D, C, G, F, E
- e. S, A, B, D, C, E, F, G

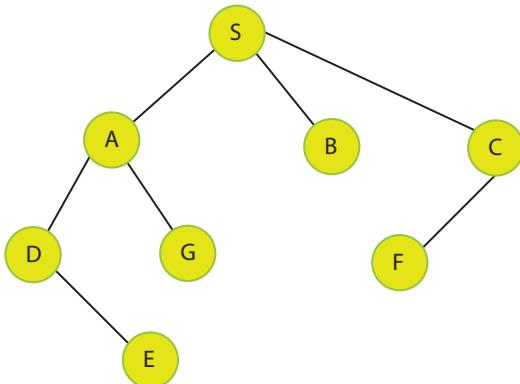
3.



Yukarıdaki çizge üzerinde, başlangıç noktası S olacak şekilde önce derinliğine arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu varsa, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir. Buna göre çizgedeki düğümlerin ziyaret edilme sırası aşağıdakilerden hangisidir?

- a. S, A, E, F, B, C, D
- b. S, A, E, F, C, B, D
- c. S, A, B, C, D, E, F
- d. S, B, D, F, C, A, E
- e. S, B, C, F, D, A, E

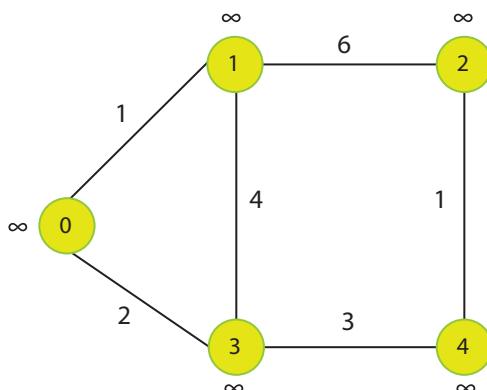
4.



Yukarıdaki çizge üzerinde, başlangıç noktası S olacak şekilde önce derinliğine arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu varsa, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir. Buna göre çizgedeki düğümlerin ziyaret edilme sırası aşağıdakilerden hangisidir?

- a. S, A, D, G, B, C, F, E
- b. S, A, B, C, D, G, F, E
- c. S, B, C, F, A, G, D, E
- d. S, A, D, E, G, B, C, F
- e. S, A, D, G, E, B, C, F

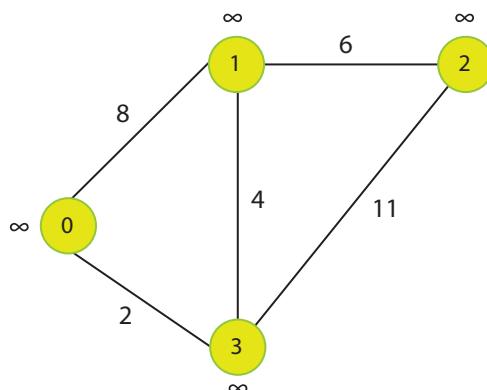
5.



Yukarıdaki çizge üzerinde, başlangıç noktası 0 düğümü olacak şekilde Dijkstra en kısa yol algoritması çalıştırılacaktır. Bu algoritmanın çalışması sonrasında 0 düğümü ile 0, 1, 2, 3, 4 düğümleri arasında hesaplanan en kısa mesafeler aşağıdaki seçeneklerden hangisinde sırasıyla doğru olarak verilmiştir?

- a. 0, 1, 2, 6, 6
- b. 0, 1, 7, 7, 1
- c. 0, 1, 7, 2, 0
- d. 0, 1, 7, 2, 5
- e. 0, 1, 6, 2, 5

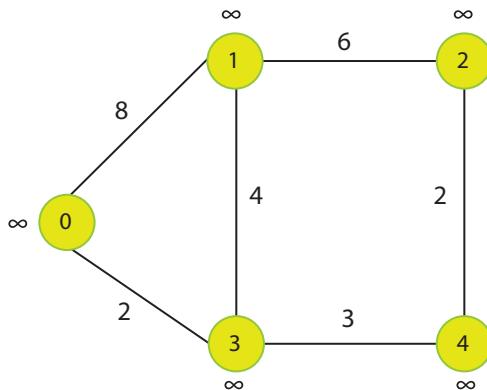
7.



Yukarıdaki çizge üzerinde, başlangıç noktası 0 olacak şekilde Dijkstra en kısa yol algoritması çalıştırılacaktır. Bu algoritmanın çalışması sonrasında 0 düğümü ile 2 düğümü arasındaki en kısa mesafeyi elde etmek için sırasıyla hangi düğümler ziyaret edilmelidir?

- a. 0, 3, 1, 0
- b. 0, 1
- c. 0, 3, 2
- d. 0, 3, 1, 2
- e. 0, 1, 2

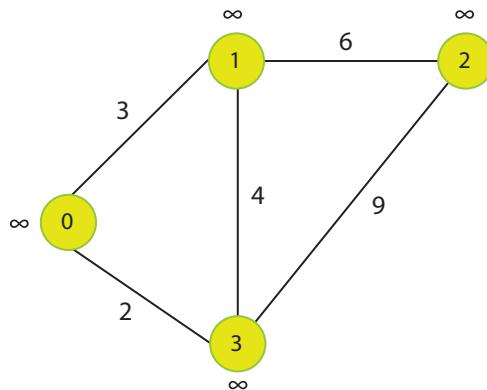
6.



Yukarıdaki çizge üzerinde, başlangıç noktası 0 düğümü olacak şekilde Dijkstra en kısa yol algoritması çalıştırılacaktır. Bu algoritmanın çalışması sonrasında 0 düğümü ile 0, 1, 2, 3, 4 düğümleri arasında hesaplanan en kısa mesafeler aşağıdaki seçeneklerden hangisinde sırasıyla doğru olarak verilmiştir?

- a. 0, 6, 7, 5, 5
- b. 0, 3, 2, 4, 5
- c. 0, 7, 6, 5, 5
- d. 0, 6, 1, 2, 5
- e. 0, 6, 7, 2, 5

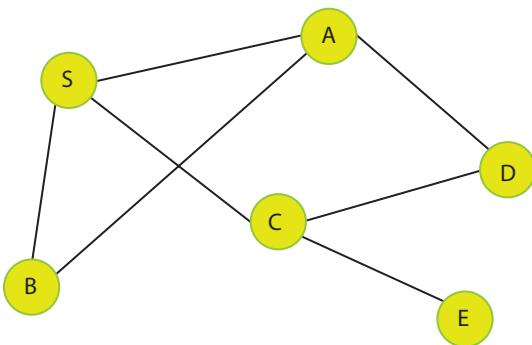
8.



Yukarıdaki çizge üzerinde, başlangıç noktası 0 olacak şekilde Dijkstra en kısa yol algoritması çalıştırılacaktır. Bu algoritmanın çalışması sonrasında 0 düğümü ile 2 düğümü arasındaki en kısa mesafeyi elde etmek için sırasıyla hangi düğümler ziyaret edilmelidir?

- a. 0, 1, 2
- b. 0, 3, 2
- c. 0, 3, 1, 2
- d. 0, 3, 1, 0, 2
- e. 0, 1, 3, 2

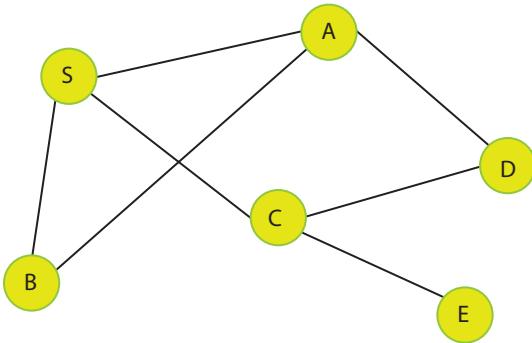
9.



Yukarıdaki çizge üzerinde, başlangıç noktası S olacak şekilde enine arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu varsa, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir. Buna göre çizgedeki düğümlerin ziyaret edilme sırası aşağıdakilerden hangisidir?

- a. S, A, D, C, E, B
- b. S, C, D, E, A, B
- c. S, A, B, C, D, E
- d. S, A, B, D, C, E
- e. S, A, B, C, E, D

10.



Yukarıdaki çizge üzerinde, başlangıç noktası S olacak şekilde önce derinliğine arama algoritması çalıştırılacaktır. Bir düğümün birden fazla komşusu varsa, bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilecektir. Buna göre çizgedeki düğümlerin ziyaret edilme sırası aşağıdakilerden hangisidir?

- a. S, A, B, C, D, E
- b. S, A, C, B, D, E
- c. S, A, B, D, C, E
- d. S, D, A, C, B, E
- e. S, C, D, E, B, A

Kendimizi Sınayalım Yanıt Anahtarı

1. e Yanınız yanlış ise “Enine Arama” konusunu yeniden gözden geçiriniz.
2. a Yanınız yanlış ise “Enine Arama” konusunu yeniden gözden geçiriniz.
3. b Yanınız yanlış ise “Önce Derinliğine Arama” konusunu yeniden gözden geçiriniz.
4. d Yanınız yanlış ise “Önce Derinliğine Arama” konusunu yeniden gözden geçiriniz.
5. e Yanınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
6. e Yanınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
7. d Yanınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
8. a Yanınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
9. c Yanınız yanlış ise “Enine Arama” konusunu yeniden gözden geçiriniz.
10. c Yanınız yanlış ise “Önce Derinliğine Arama” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

İstenilen değişiklikler yapıldıktan sonra `enine_arama.c` programının kodları içerisindeki `main` fonksiyonunun içeriği aşağıdaki gibi olmalıdır. Ayrıca, bu örnekteki düğüm sayısı 9 olduğundan programın en başında tanımlanan ve değeri 5 olan `MAKSIMUM` değişkeninin değeri 9 olarak güncellenmelidir.

```
void main()
{
    int i, j;
    // komsuluk matrisini baslangicta sifirla
    for (i = 0; i < MAKSIMUM; i++)
    {
        for (j = 0; j < MAKSIMUM; j++)
            komsulukmatrisi[i][j] = 0;
    }
    // dugumleri ekle
    dugum_ekle('S'); // 0
    dugum_ekle('A'); // 1
    dugum_ekle('B'); // 2
    dugum_ekle('C'); // 3
    dugum_ekle('D'); // 4
    dugum_ekle('E'); // 5
    dugum_ekle('F'); // 6
    dugum_ekle('G'); // 7
    dugum_ekle('H'); // 8

    // kenar baglantilarini ekle
    yonsuz_kenarbaglantisi_ekle(0, 1); // S - A
    yonsuz_kenarbaglantisi_ekle(0, 3); // S - C
    yonsuz_kenarbaglantisi_ekle(0, 8); // S - H
    yonsuz_kenarbaglantisi_ekle(1, 7); // A - G
    yonsuz_kenarbaglantisi_ekle(3, 2); // C - B
    yonsuz_kenarbaglantisi_ekle(3, 4); // C - D
    yonsuz_kenarbaglantisi_ekle(8, 4); // H - D
    yonsuz_kenarbaglantisi_ekle(2, 7); // B - G
    yonsuz_kenarbaglantisi_ekle(2, 5); // B - E
    yonsuz_kenarbaglantisi_ekle(5, 6); // E - F

    printf("\nEnine Arama Sonuclari: ");
    enine_arama_algoritmasi();
    getch();
}
```

Sıra Sizde 2

İstenilen değişiklikler yapıldıktan sonra `once_derinligine_arama.c` programının kodları içerisindeki `main` fonksiyonun içeriği aşağıdaki gibi olmalıdır. Ayrıca, bu örnekteki düğüm sayısı 10 olduğundan programın en başında tanımlanan ve değeri 5 olan `MAKSIMUM` değişkeninin değeri 10 olarak güncellenmelidir.

```
void main()
{
    int i, j;

    // komsuluk matrisini baslangicta sifirla
    for (i = 0; i<MAKSIMUM; i++)
    {
        for (j = 0; j<MAKSIMUM; j++)
            komsulukmatrisi[i][j] = 0;
    }

    dugum_ekle('S');      // 0
    dugum_ekle('B');      // 1
    dugum_ekle('C');      // 2
    dugum_ekle('D');      // 3
    dugum_ekle('E');      // 4
    dugum_ekle('F');      // 5
    dugum_ekle('G');      // 6
    dugum_ekle('H');      // 7
    dugum_ekle('I');      // 8
    dugum_ekle('J');      // 9

    yonsuz_kenarbaglantisi_ekle(0, 1);      // S - B
    yonsuz_kenarbaglantisi_ekle(0, 2);      // S - C
    yonsuz_kenarbaglantisi_ekle(1, 3);      // B - D
    yonsuz_kenarbaglantisi_ekle(1, 4);      // B - E
    yonsuz_kenarbaglantisi_ekle(3, 7);      // D - H
    yonsuz_kenarbaglantisi_ekle(4, 8);      // E - I
    yonsuz_kenarbaglantisi_ekle(4, 9);      // E - J
    yonsuz_kenarbaglantisi_ekle(2, 5);      // C - F
    yonsuz_kenarbaglantisi_ekle(2, 6);      // C - G

    printf("\nOnce Derinligine Arama Sonuclari: ");
    once_derinligine_arama_algoritmasi();
    getch();
}
```

Sıra Sizde 3

İstenilen değişiklikler yapıldıktan sonra dijkstra_en_kisa_yol.c programının kodları içerisindeki main fonksiyonunun içeriği aşağıdaki gibi olmalıdır. Ayrıca, bu örnekteki düğüm sayısı 8 olduğundan programın en başında tanımlanan ve değeri 5 olan MAKS değişkeninin değeri 8 olarak güncellenmelidir.

```
void main()
{
    int i, j, baslangic_dugum;
    // komsuluk matrisini baslangicta sifirla
    for (i = 0; i<MAKS; i++)
    {
        for (j = 0; j<MAKS; j++)
            komsulukmatrisi[i][j] = SONSUZ;
    }
    // dugumleri ekle
    dugum_ekle('0');    // 0
    dugum_ekle('1');    // 1
    dugum_ekle('2');    // 2
    dugum_ekle('3');    // 3
    dugum_ekle('4');    // 4
    dugum_ekle('5');    // 5
    dugum_ekle('6');    // 6
    dugum_ekle('7');    // 7

    // kenar baglantilarini ekle
    yonsuz_kenarbaglantisi_ekle(0, 1, 8);      // 0 - 1 = 8
    yonsuz_kenarbaglantisi_ekle(0, 2, 2);      // 0 - 2 = 2
    yonsuz_kenarbaglantisi_ekle(0, 3, 5);      // 0 - 3 = 5
    yonsuz_kenarbaglantisi_ekle(1, 3, 2);      // 1 - 3 = 2
    yonsuz_kenarbaglantisi_ekle(1, 5, 13);     // 1 - 5 = 13
    yonsuz_kenarbaglantisi_ekle(2, 3, 2);      // 2 - 3 = 2
    yonsuz_kenarbaglantisi_ekle(2, 4, 5);      // 2 - 4 = 5
    yonsuz_kenarbaglantisi_ekle(3, 4, 1);      // 3 - 4 = 1
    yonsuz_kenarbaglantisi_ekle(3, 5, 6);      // 3 - 5 = 6
    yonsuz_kenarbaglantisi_ekle(3, 6, 3);      // 3 - 6 = 3
    yonsuz_kenarbaglantisi_ekle(4, 6, 1);      // 4 - 6 = 1
    yonsuz_kenarbaglantisi_ekle(5, 6, 2);      // 5 - 6 = 2
    yonsuz_kenarbaglantisi_ekle(5, 7, 3);      // 5 - 7 = 3
    yonsuz_kenarbaglantisi_ekle(6, 7, 6);      // 6 - 7 = 6

    // Baslangic dugumunu gir
    printf("\nBaslangic dugumunu sec: ");
    scanf("%d", &baslangic_dugum);

    // Dijkstra algoritmasını calistir
    dijkstra(baslangic_dugum);
    getch();
}
```

Yararlanılan ve Başvurulabilecek Kaynaklar

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

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

Weiss M.A. (2013) **Data Structures and Algorithm Analysis in C++**, 4th Edition, Pearson.