



ANADOLU ÜNİVERSİTESİ



anadolulum

eKampus

dilediğin yerden,
dilediğin zaman,
öğrenme fırsatı!



Çıkmış Sınav Soruları



Ünite Özeti



Ders Kitabı (PDF)



Etkileşimli eKitap



Yaprak Test



Sesli Kitap
Sesli Özet



eSeminer



eKantin



1 Soru
1 Cevap



Deneme Sınavı



Tartışma Forumu



444 10 26

0850 200 46 10-19 (10hat)

www.anadolu.edu.tr

ekampus.anadolu.edu.tr

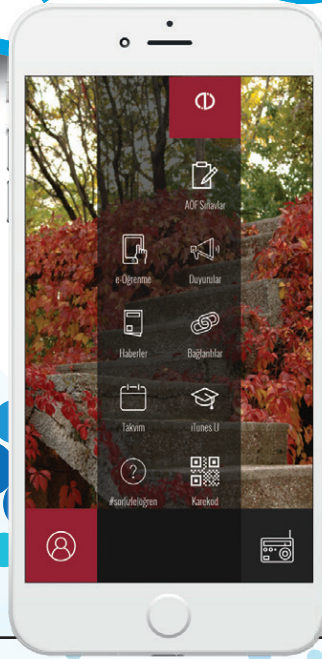
[/anadoluuniversitesi](https://www.facebook.com/anadoluuniversitesi) [/Anadolu_Univ](https://www.instagram.com/anadoluuniv) [instagram.com/anadoluuniv](https://www.instagram.com/anadoluuniv)



ANADOLU ÜNİVERSİTESİ

ANADOLU

mobil



- **AÖF sınavları**
- **sınav giriş belgesi**
- **sınav sonuçları**
- **itunes u**
- **duyurular**
- **sor/izle/öğren**
- **ders kitabı**
- **sesli kitap**
- **deneme sınavları**
- **konu anlatım videoları**
- **TRT okul videoları**
- **e-seminer dersleri**
- **takvim**



444 10 26

0850 200 46 10-19 (10hat)

www.anadolu.edu.tr

ekampus.anadolu.edu.tr



ANADOLU ÜNİVERSİTESİ

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)
Yrd.Doç.Dr. Sevcan YILMAZ GÜNDÜZ (Ünite 4, 5)
Yrd.Doç.Dr. Alper Kürşat UYSAL (Ünite 6, 7, 8)

Editör

Doç.Dr. Serkan GÜNAL

Bandrol Uygulamasına İlişkin Usul ve Esaslar Hakkında
Yönetmeliğin 5 inci Maddesinin İkinci Fıkrası Çerçevesinde
Bandrol Taşınması Zorunlu Değildir.

Bu kitabın basım, yayım ve satış hakları Anadolu Üniversitesine aittir.
“Uzaktan Öğretim” tekniğine uygun olarak hazırlanan bu kitabın bütün hakları saklıdır.
İlgili kuruluştan izin almadan kitabın tümü ya da bölümleri 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.

UZAKTAN ÖĞRETİM TASARIM BİRİMİ

Genel Koordinatör

Prof.Dr. Müjgan Yazıcı

Genel Koordinatör Yardımcısı

Doç.Dr. İrem Erdem Aydın

Öğretim Tasarımcısı

Prof.Dr. T. Volkan Yüzer

Grafik Tasarım Yönetmenleri

Prof. Tevfik Fikret Uçar

Yrd.Doç. Nilgün Salur

Öğr.Gör. Cemalettin Yıldız

Kitap Yazım Basım ve Dağıtım Koordinatörü

Uzm. Nermin Özgür

Kapak Düzeni

Doç.Dr. Halit Turgay Ünalın

Grafikerler

Ayşegül Dibek

Özlem Çayırılı

Hilal Küçükdağışan

Dizgi

Açıköğretim Fakültesi Dizgi Ekibi

Algoritmalar ve Programlama

ISBN

978-975-06-2060-7

1. Baskı

Bu kitap ANADOLU ÜNİVERSİTESİ Basımevinde 10.000 adet basılmıştır.
ESKİŞEHİR, Aralık 2016

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	
Akış 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 Anahtarı	14	
Sıra Sizde Yanıt Anahtarı	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	
Bağlı Listeler ile Dizilerin Karşılaştırması	22	
Bağlı Liste Türleri	22	
Tek Yönlü Bağlı Liste	22	
Çift Yönlü Bağlı Liste	23	
Dairesel Bağlı Liste	23	
Bağlı 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	
Bağlı Liste ile Kuyruk Uygulaması	28	
YIĞINLAR	30	

Dizi ile Yiğın Uygulaması	31
Bağlı Liste ile Yiğın 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, Yiğın Ağaçları ve Özetleme Tabloları.....	40
GİRİŞ	41
AĞAÇLAR	41
İkili Ağaçlar (Binary Trees)	42
İkili Ağaçlarda 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 Korunumu	49
AVL Ağacından Düğüm Çıkarma ve Denge Korunumu	50
YIĞIN AĞAÇLARI	51
Dizi ile Yiğın Ağacı Uygulaması	52
Yiğın Ağacında En Küçük Elemanı Elde Etme	52
Yiğın Ağacından En Küçük Elemanı Çıkarma: Aşağı Yönlendirme	52
Yiğın 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 Sınyalım	80
Kendimizi Sınyalım 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
ASİMPOTİK 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 Sınyalım	99
Kendimizi Sınyalım 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 Sınyalım	113
Kendimizi Sınyalım 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 Sınyalım	141
Kendimizi Sınyalım 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 işlemlerin 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 faydalanabiliriz. Algoritmanın bir bilgisayar tarafından gerçekleştirilmesi 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 saklanmasına 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 yığınlar ikinci ünite; ağaçlar, yığın ağaçları ve özetleme tabloları ise üçüncü ünite açıklanmıştır. Dördüncü ünite, algoritma tasarımı 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 özyinelemesiz algoritmaların analizi konuları anlatılmıştır. Altıncı ünite, programlamada sıklıkla ihtiyaç duyulan arama işlemleri için kullanılan başlıca arama algoritmaları açıklanmıştır. Yedinci ünite, sıralama işlemleri için faydalanılan 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 gerçekleştirilmesi için C programlama dili temel alınmış ve programlama örnekleri bu doğrultuda hazırlanmıştır. Her bir ünite sunulan bilgiler, açıklayıcı örneklerle desteklenmiş, konulara olan hâkimiyetinizin artmasına yönelik “Sıra Sizde” çalışmaları sunulmuş, ünite sonunda öğrendiklerinizi sınavınıza imkân tanıyan “Kendimizi Sınavalım” sorularına yer verilmiştir.

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

Editör
Doç.Dr. Serkan GÜNAL

1

Amaçlarımız

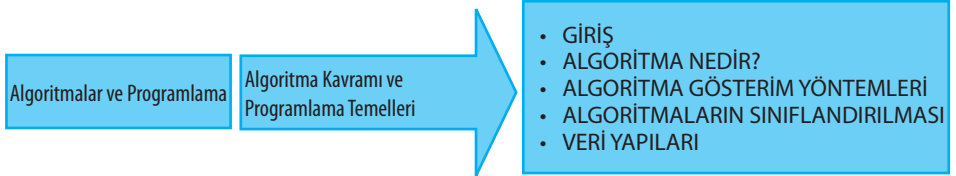
Bu üniteyi tamamladıktan sonra;

- 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



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 belirmesi 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 bilmediğimiz bir restorana bulmamıza yardımcı olan yol tarifi, bir elektronik cihazın kullanım kılavuzu, algoritmaların günlük hayatımızdaki kullanımına örnek olarak gösterilebilir.

Günlük yaşantımızda 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 önündeki 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 çıkılır.

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



Çamaşır makinesi kullanarak çamaşırları yıkamak, günlük hayatta 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:* Algoritmalarda girdi ve çıktı bilgileri olmalıdır. Girdi bilgisi algoritmaya 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 tasarımına 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

*Euclid
Algoritmasının
Konuşma Dili ile
Gösterimi.*

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

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

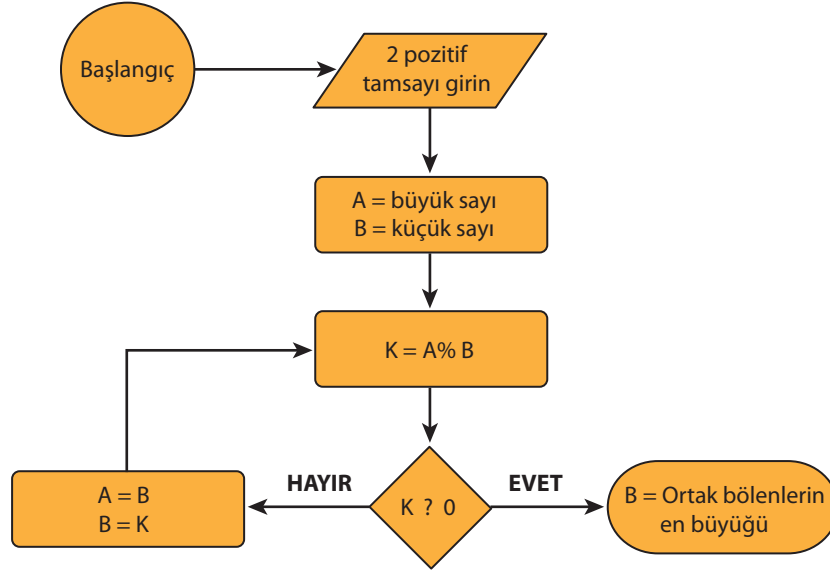
Akış şeması, algoritmaların gösteriminde kullanılan faydalı bir yöntemdir. Bir akış şemasında algoritma adımlarını ifade eden kutucuklar, adımlar arası geçişleri gösteren oklar, karar verme mekanizmaları olarak kullanılan şekiller bulunabilir.

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 Akış Şeması ile Gösterimi.



İNETERNET



Web tarayıcıları üzerinden çevrimiçi olarak kullanılabilen (ücretsiz veya deneme amaçlı) çizim yazılımlarından bazılarında 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 harmanlanmasından 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 sonlandırıldığı 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 satırdır. Bu adımda algoritmanın tamamlandığı 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 yöntemlere göre sınıflandırılabilir. Algoritmaları sınıflandırmadaki temel amaç, problemlerin çözümünde başvurulabilecek değişik metotları ve alternatifleri tespit edebilmektir.

Ünitemizin bu bölümünde, aşağıda listelenen algoritma türleri incelenecektir:

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

Özyinelemeli Algoritmalar (Simple Recursive Algorithms)

Kendisini doğrudan veya dolaylı olarak çağıran algoritmalara özyinelemeli algoritma adı verilir. Bu algoritmalarda, 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 hesaplamaya 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 algoritmalarlardır. 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 nesnelere vardır. Sırt çantasına hangi nesnelere doldurulduğunda, çantaya konan nesnelere toplam değeri en fazla olur?

SIRA SİZDE



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

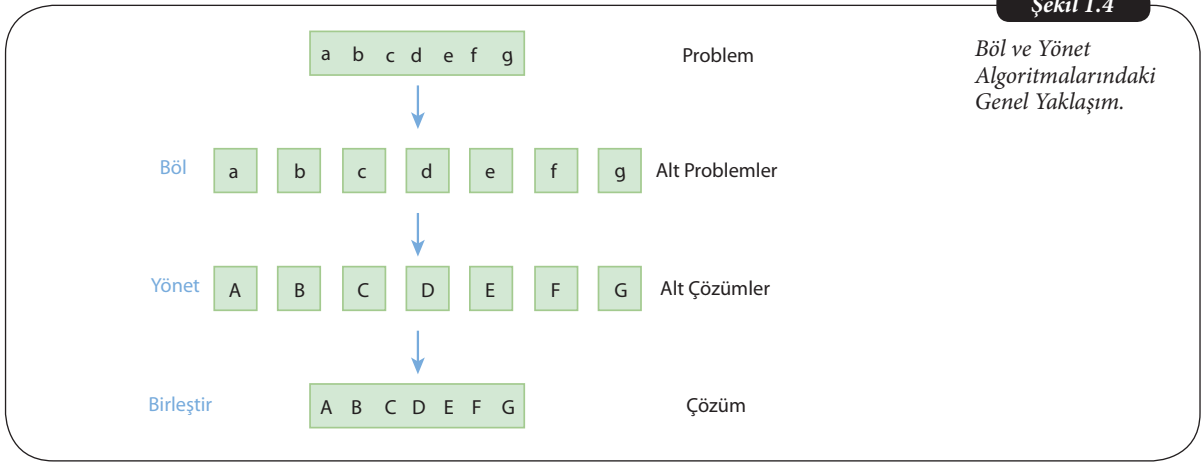
Böl ve Yönet Algoritmaları (Divide and Conquer Algorithms)

Böl ve yönet algoritmaları, problemlere mümkün olan en küçük alt parçalara ayrıldığı, her bir alt parçanın diğerlerinden bağımsız şekilde çözüldüğü algoritmalarlardır. 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 bölünemeyecek 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 çözümlere, 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 çözdüğümüz 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 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)

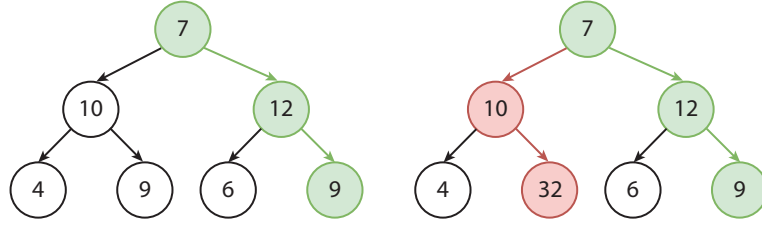
Bir problem için mümkün olan en doğru çözümü hedefleyen algoritmalar açgözlü algoritmalar adı verilir. Açgözlü algoritmalar yerel olarak optimum sonuç elde edilirken, bulunan sonuç her zaman için en iyi çözüme karşılık gelmeyebilir.

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 şehre 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.

Şekil 1.5

En Fazla Yol Kat Etme Probleminin Açgözlü Algoritma ile Çözümü.



a. Aç gözlü algoritma ile en iyi çözüm

b. Aç gözlü 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 algoritmalarda çok fazla işlem yapılır ve çözüm yolu optimumdan uzaktır. Problemdaki veri hacmi büyüdükçe, 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

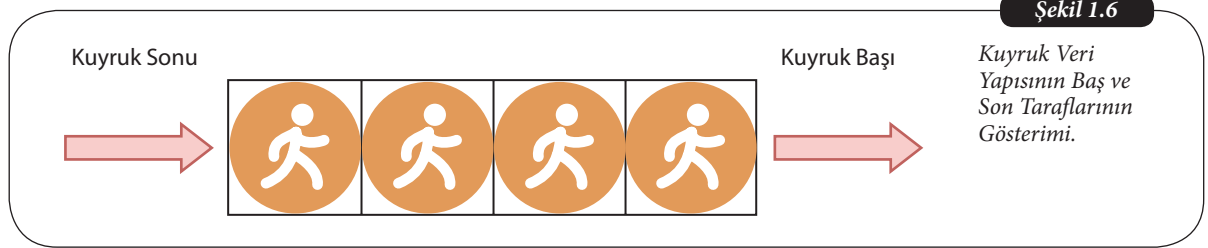
Bir tamsayı dizisinde belirli bir tamsayıyı arayacak kaba kuvvet algoritması kodlayınız. Gelecekteki algoritma, aranan eleman dizide bulunursa 1 değerini, bulunamazsa 0 değerini çıktı olarak vermelidir.

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, içerdiği 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 incelenecek ve örneklenecek veri yapıları şunlardır:

- Diziler (Arrays)
- Bağlı Listeler (Linked Lists)
- Kuyruklar (Queues)
- Yığınlar (Stacks)
- Ağaçlar (Trees)
- Yığın Ağaçları (Heaps)
- Özetleme Tabloları (Hash Tables)
- Çizgeler (Graphs)

Özet



Programlamanın 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 yapılmasında 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, okunurluğunu ve anlaşılabilirliğini arttıran 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 oluşturulurken 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 İzlemeli Algoritmalar, Böl ve Yönet Algoritmaları, Dinamik Programlama, Ağaçlı Algoritmalar ve Kaba Kuvvet Algoritmaları incelenmiştir. Özyinelemeli Algoritma-

lar, kendisini doğrudan veya dolaylı olarak çağıran algoritmalar. Problemler daha küçük parçalara indirgenir ve bu parçalar için oluşturulan çözümler birleştirilerek ana problemin çözümü elde edilir. Geri İzlemeli Algoritmalar, problemin çözüm aşamasında ilerlerken, olası çözüm yollarının hepsini deneyerek bir sonraki adıma geçmeye çalışan algoritmalar. Denenen çözüm yolundan sonuç 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 algoritmalar. 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. Ağaçlı Algoritmalar, bir problem için mümkün olan en doğru çözümü hedefleyen algoritmalar. Kaba Kuvvet Algoritmaları, bir problemin çözüm aşamasında, yeterli bir çözüm elde edene kadar tüm olasılıkları deneyen algoritmalar.



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, içerdiği elemanların mantıksal düzeni ve elemanlar üzerinde yapılabilecek işlemler ile tanımlanır.

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

Kendimizi Sıyalım

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

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

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

- Verimlilik
- Açıklık
- Sonluluk
- Genellik
- Doğruluk

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

- Sonluluk
- Girdi ve çıktı bilgisi
- Açıklık
- Genellik
- Verimlilik

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

- Konuşma dili
- Sözde kod
- Kaynak kodu
- Akış şeması
- 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

```

- İki tamsayıdan büyük olanı bulmak
- Pozitif doğal sayılarda toplama işlemi yapmak
- Bir tamsayı kümesinin eleman sayısını hesaplamak
- Bir tamsayı kümesinin en küçük elemanını bulmak
- İki tamsayının 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?

- Özyinelemeli algoritmalar
- Kaba kuvvet algoritmaları
- Açgözlü algoritmalar
- Böl ve yönet algoritmaları
- Geri izlemeli algoritmalar

7. Bir çelik kasanın kilidini açmak için olası tüm şifreleri teker teker deneyen kişinin kullandığı yöntem, hangi algoritma sınıfına girer?

- Kaba kuvvet algoritmaları
- Açgözlü algoritmalar
- Geri izlemeli algoritmalar
- Böl ve yönet algoritmaları
- Dinamik programlama

8. Dinamik programlamada, bir problemin alt kümelerinin çözümlerini tekrar tekrar hesaplamak yerine bilgisayar hafızasında saklayan yöntem ne ad verilir?

- Programlama
- Sorgulama
- Denetleme
- Hesaplama
- Ezberleme

9. Bilgisayar programlarında verilerin sistematik ve organize bir şekilde saklanmasını sağlayan kavram hangisidir?

- Programlama dili
- Derleyici
- Veri yapısı
- Ağ bağlantısı
- Monitör

10. Aşağıdakilerden hangisi bir veri yapısı türü **değildir**?

- Yığın
- Gösterici
- Kuyruk
- Bağlı liste
- Ağaç

Kendimizi Sınavalım Yanıt Anahtarı

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

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

Programlamanın 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 çamaşırın yıkanmasını ve temizlenmesini tarif eder. Bu algoritmanın olası adımları aşağıda belirtilmiştir:

- Çamaşır makinesinin kapağı açılır.
- Yıkanacak çamaşır makineye doldurulur.
- Çamaşır makinesinin kapağı kapatılır.
- Çamaşıra uygun temizlik maddeleri (deterjan, yumuşatıcı, leke çıkarıcı, vs.), deterjan haznesine doldurulur.
- Uygun yıkama ayarları (yıkama programı, sıcaklık, sıklık, 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 içerisine 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 sayıyı 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 ermekte ve sonraki elemanların kontrol edilmesine gerek kalmamaktadır. Aranan değer, dizinin hiçbir elemanının değeri ile aynı değilse, algoritma 0 değeri döndürerek sona ermektedir.

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

int tamsayilar[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 == tamsayilar[i]) {
            return 1;
        }
    }

    return 0;
}

int main(void) {
    int n;

    for(n=1; n<50; n++) {
        if(search(n) == 1) {
            printf("%d dizide vardir.\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

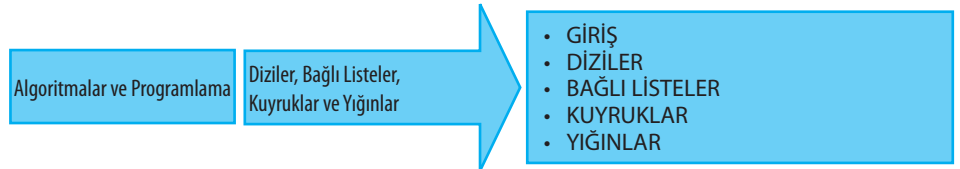
Bu üniteyi tamamladıktan sonra;

- Tek boyutlu dizi tanımlamayı ve dizinin elemanlarına değer atamayı ifade edebilecek,
- Çok boyutlu dizi yapılarını açıklayabilecek,
- Bağlı liste yapısını tanımlayabilecek ve bağlı liste çeşitlerini sıralayabilecek,
- Kuyruk yapısının mantığını açıklayacak ve temel kuyruk işlemlerini sıralayabilecek,
- Dizilerle ve bağlı listelerle kuyruk veri yapısını oluşturabilecek,
- Yığın yapısının mantığını kavrayacak ve temel yığın işlemlerini sıralayabilecek,
- Dizilerle ve bağlı listelerle yığın veri yapısını oluşturabilecek bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

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

İçindekiler



Diziler, Bağlı Listeler, Kuyruklar ve Yığınlar

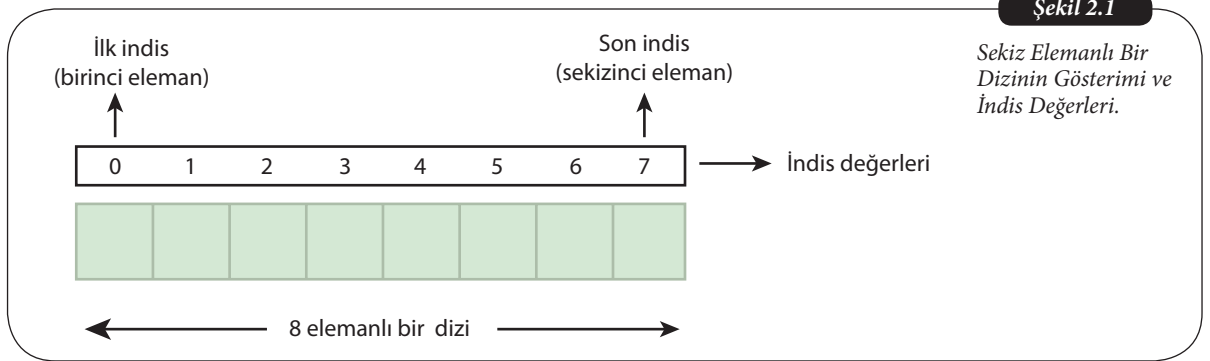
GİRİŞ

Kitabımızın bu ünitesinde işlenen konuların ana başlıkları, 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 eksiğine 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 değinilecektir.



DİKKAT

Dizilerin Tanımlanması

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

Şekil 2.2

Dizilerin
Tanımlanmasındaki
Genel İfade.

```
<dizi-tipi> <dizi-adi> [ dizi-uzunluđu ];
```

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-adi:** Tanımlanan dizinin adını ifade eder.
- c. **dizi-uzunluđu:** Köşeli parantez içerisinde belirtilen bu değer, dizinin uzunluğunu belirtir.

ÖRNEK 2.1

int ve char tiplerinde tek boyutlu dizi tanımlamaları.

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

Örnek 2.1'deki kod satırlarında iki farklı dizi tanımlaması yapılmıştır. İlk dizi int veri tipinde, tamsayiDizisi 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 ayırır; fakat dizi elemanlarına bir değer ataması yapılmaz.

Dizilere değer atamak 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, programcının ihtiyacı doğrultusunda en doğru yöntemin uygulanmasıdır.

ÖRNEK 2.2

Döngü kullanımı ile dizi elemanlarının sıfırlanması.

```
#include <stdio.h>

int main(void) {
    int dizi[6],
        int 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, tamsayiDizisi 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 tanımlanmasında, dizinin boyutu derleyici tarafından algılanır ve hatasız çalıştırılır.

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



1

SIRA SİZDE

Çok Boyutlu Diziler

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

İki 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

İki Boyutlu ($4 \times 3 = 4$ Satır, 3 Sütun) Bir Dizin 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 atanan değerleri ekrana yazdırma işlemlerini gerçekleştiren program, Örnek 2.4'te verilmiştir.

ÖRNEK 2.4

İki 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 hakkındaki deneyiminizin artmasına katkıda bulunacaktır.



DİKKAT

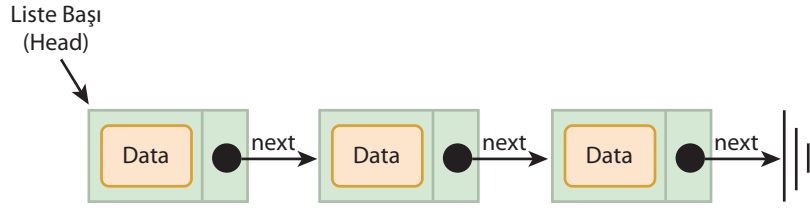
BAĞLI LİSTELER

Bağlı liste (linked list), aynı türden nesnelere 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 bağlanmışlardır. 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 Listenin Mantıksal Gösterimi.



Bağlı Listeler ile Dizilerin Karşılaştırması

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

- *Veri yapısı uzunluğu:* Dizilerde veri yapısının uzunluğu sabittir, gerekli durumlarda dizi uzunluğu arttırılmaz veya azaltılmaz. Bağlı listelerde uzunluk dinamiktir, yeni nesnelere eklenebilir, var olan nesnelere silinebilir.
- *Hafıza kullanımı:* Bağlı listelerdeki her bir nesnenin göstericisi için, bilgisayar hafızasında yer ayrılması gerekir. 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 gerekir.

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 ortaya çıkmasına neden olur. Bağlı listelerdeki üç tür aşağıda listelenmiştir:

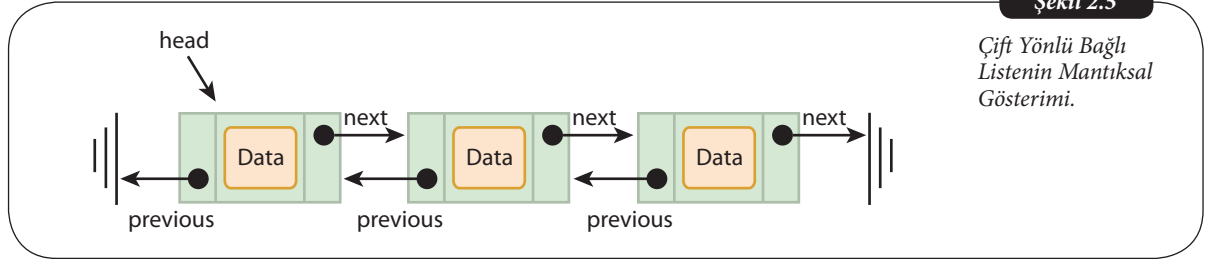
- Tek yönlü bağlı liste (Singly linked list)
- Çift yönlü bağlı liste (Doubly linked list)
- Dairesel bağlı liste (Circular linked list)

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

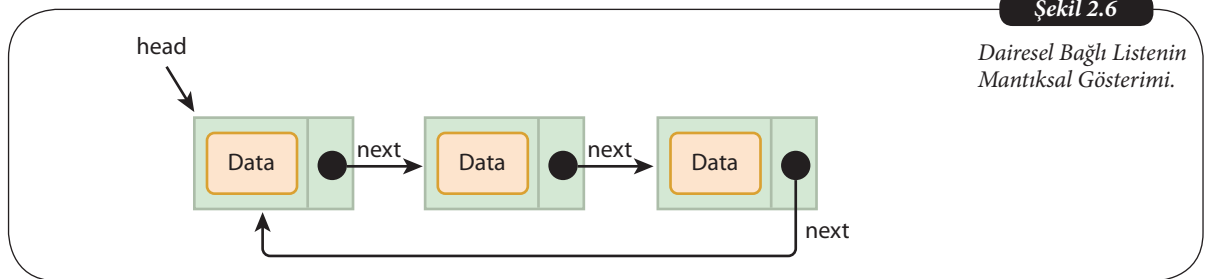
Çift yönlü bağlı listelerde, liste düğümleri arasında hem ileriye hem de geriye doğru gidilebilir. Çift yönlü bağlı listenin bir düğümü, bir sonraki düğümü işaret eden göstericinin (next pointer) yanı sıra, bir önceki düğümü işaret eden göstericiyi (previous pointer) de içerir. Çift yönlü bağlı liste 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 dairesele hale gelmiş olur. Bağlı listelerin bu çeşidine dairesele bağlı liste denilmektedir. Dairesel bağlı liste için bir örnek, Şekil 2.6'da gösterilmiştir.

Gerek tek yönlü gerek çift yönlü bağlı listelerin dairesele hale getirilmesi mümkündür. Bazı kaynaklarda, dairesele bağlı listeler, kendi içlerinde 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 değişiklik 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 bölgesine 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 bölgesinden eleman çıkarılabilir. Çıkarma işleminde çıkarılacak düğüm, free fonksiyonu ile hafızadan silinir. Çıkarma işlemi listenin ara bir bölgesinden yapılacaksa, öncelikle çıkarılacak düğüm listede içerisinde bulunmalı, düğüm bulunduğundan sonra çıkarma işlemi gerçekleştirilmelidir. Bağlı listelerde çıkarma işlemini baş taraftan gerçekleştiren fonksiyon, Örnek 2.8'de yer almaktadır.

ÖRNEK 2.8*Bağlı listede liste başından eleman çıkarma.*

```

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 ulaşıncaya kadar, yani bir sonraki düğümü işaret eden göstericisi (next pointer) NULL olan eleman bulunana kadar devam eder. Listenin elemanları gezinilirken, listede eleman arama veya elemanları ekrana yazdırma gibi işlemler de yapılabilir. Örnek 2.9'da listenin elemanlarını gezinen 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 adımında i değeri listenin başına eklenmiştir. Daha sonra, 1'den 4'e kadar çalışacak yeni bir döngü kurulmuş ve döngünün her adımında 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 denemeniz, konuları özümsemeye katkı sağlayacaktır.

KUYRUKLAR

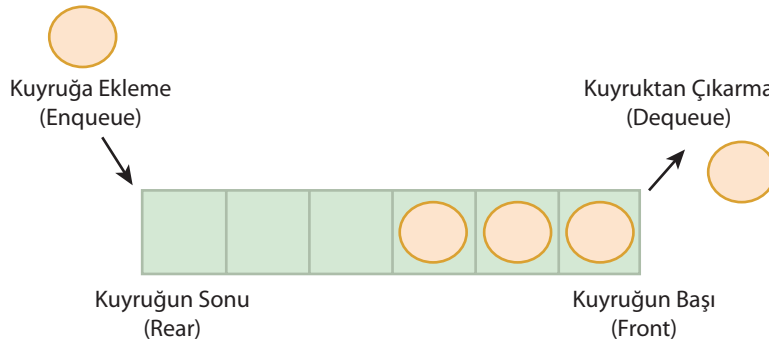
Günlük yaşantımızda kuyruğa girmek ve sıra beklemek oldukça rutin bir eylemdir. Bankalarda, gişelerde, süpermarketlerde, trafikte vb. birçok alanda insanlar sıralar oluşturur, işini bitiren kişiler kuyruktan ayrılır, yeni gelen kişiler kuyruğa dahil olur. Günlük hayattaki bu kuyruk mantığı, programlamada 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 işlemler olan ekleme (enqueue) son taraftan, çıkarma (dequeue) ise baş taraftan gerçekleştirilir. Dolayısıyla kuyruğa ilk giren eleman, kuyruktan ilk çıkan eleman olur (Şekil 2.7).

FIFO (First-In First-Out): Programlamada kuyruklar FIFO kuralı ile anılır. Bu ifade, "İlk Giren İlk Çıkar" şeklinde tercüme edilebilir.

Şekil 2.7

Kuyruk Veri Yapısının Mantıksal Gösterimi.



Kuyruk mantığının bir veri yapısı olarak programlanmasında 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 faydalanılabilir. 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 tamsayıya ih-

tiyaç duyulur. Kuyruk veri yapısının tamsayı tipinde değerleri saklayacağını varsayarsak, 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("Kuyrukta 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("Kuyrukta 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. Kuyrukta tamsayı verileri saklayacak diziye `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 çıkarma yapmak için `dequeue()` fonksiyonu tanımlanmıştır.

Ekleme işleminde öncelikle kuyruğun doluluğu (dizinin uzunluğu olan `N` ile `count` değişkenini karşılaştırarak) kontrol edilir ve kuyrukta yer varsa işleme devam edilir. Kuyruğa eklenecek `a` sayısı, `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 kuyrukta herhangi bir eleman varsa işleme devam edilir. Kuyruktan çıkarılacak değer, `queue` dizisinin `front` ile belirtilen indisinde yer almaktadır. Çıkarma işlemiyle `front` değeri 1 arttırılırken, dizinin eleman sayısı 1 azaltılır. Örnekte, kuyruktan çıkarılan değer `a` sayısına atanarak ekrana yazdırılmıştır, fakat ekrana yazdırma dışında, bu değer kullanılarak farklı işlemler de gerçekleştirilebilir.

Ekleme ve çıkarma işlemlerinde önemli bir ayrıntı, *queue* dizisinde dairesel bir yapı kurulmasıdır. Eklemede *rear* değişkeni, çıkarmada 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

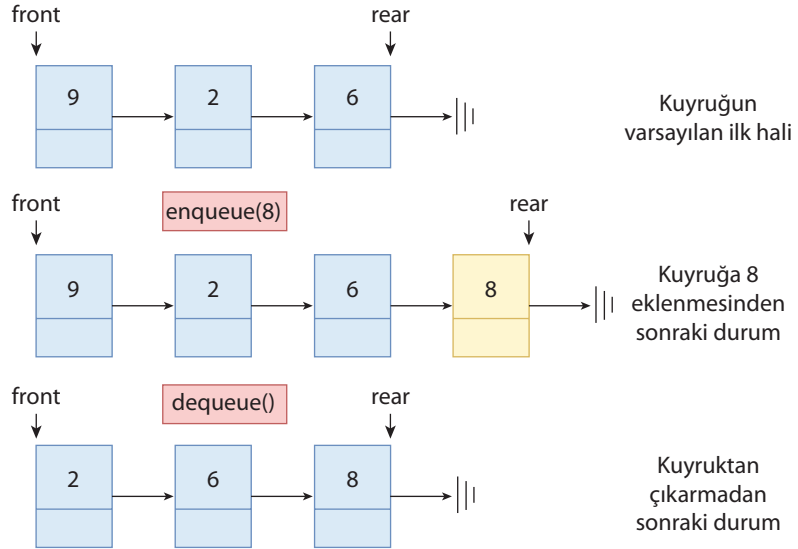
Örnek 2.11'i temel alarak, bir kuyruk uygulaması geliştirin. Yazacağınız programda kuyruğa sırasıyla 5, 12, 9, 8, 0, 1, 7 sayılarını ekleyin, kuyruktan çıkarma işlemini 2 defa uygulayın, kuyruğa tekrar 5 sayısını ekleyin.

Bağlı Liste ile Kuyruk Uygulaması

Kuyruk yapısını programlamak için bir bağlı listeden de faydalanılabilir. Bu yöntemde bağlı listenin elemanlarını oluşturacak bir veri yapısına ve kuyruğun başını ve sonunu takip edecek göstericilere 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 sayısını içeren eleman kuyruğun başında, 6 sayısını içeren eleman ise kuyruğun sonundadır. Tanımlanan bu kuyruğa, *enqueue* işlemi ile 8 sayısını içeren bir eleman eklenmiştir ve kuyruğun sonu, bu elemanı gösterecek şekilde güncellenmiştir. Daha sonraki aşamada, *dequeue* ile kuyruğun başındaki eleman kuyruktan çıkarılmıştır ve kuyruğun başı, 2 sayısını içeren elemanı gösterecek şekilde güncellenmiştir.

Bağlı liste kullanımı ile kuyruk uygulamasında tamsayı tipinde değerlerin saklanacağını varsayarsak, 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 kuyruğa 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* göstericisinden 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 çıkarma yapmak için *dequeue()* fonksiyonu tanımlanmıştır.

Ekleme işleminde *malloc* fonksiyonu ile yeni eleman (*t* adı verilen *Node* gösterici) için hafızada yer açılır. Eklenecek a tamsayısı, *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 işleme 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 birden fazla eleman varsa, kuyruğun başı, kuyruğun en baştan ikinci elemanı olacak şekilde tanımlama yapılır. *free* fonksiyonu ile kuyruğun başındaki eleman, hafızadan kaldırılır.

SIRA SİZDE



3

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

DİKKAT



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

YIĞINLAR

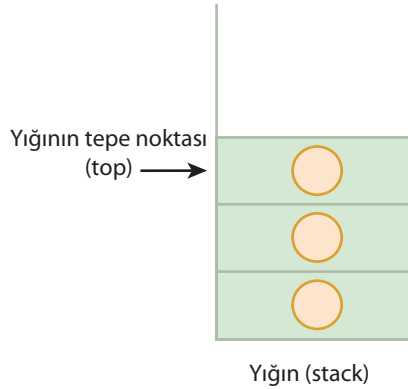
Yaşamımızdaki çeşitli aktivitelerde nesnelerin üst üste dizilmesi gerekir. Üniversite yemekhanesindeki tepsiler, restoran mutfağındaki tabaklar, elbise dolabı rafındaki kıyafetler, nesnelerin günlük yaşamda üst üste dizilmesi için gösterilebilecek basit örneklerdir. Bir üniversite yemekhanesindeki tepsilerden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

Yığın, verilerin doğrusal bir şekilde tutulduğu, ekleme ve çıkarma işlemlerinin en üst noktadan yapıldığı bir veri yapısıdır. Eklenen veri, yığının en üst noktasında saklanırken; çıkarılan veri de yığının en üst noktasından alınır. Yığının en üst noktasının takibi, yığının tepe noktası (*top*) aracılığıyla sağlanır (Şekil 2.9).

LIFO (Last-In First-Out): Programlamada yığınlar LIFO kuralı ile anılır. Bu ifade, "Son Giren İlk Çıkar" şeklinde tercüme edilebilir.

Şekil 2.9

Yığın Veri Yapısının Mantıksal Gösterimi.



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

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

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

- i. Dizilerin kullanımı ile yığın programlama
- ii. Bağlı listelerin kullanımı ile yığın programlama

Dizi ile Yığın Uygulaması

Yığın veri yapısını programlamada bir diziden faydalanılabilir. Bu yöntemde verileri tutacak bir diziyi ve yığının tepe noktasını takip edecek bir tamsayıya ihtiyaç vardır. Yığın veri yapısının tamsayı tipinde değerleri saklayacağını varsayarsak, programda gerekli olan değişkenleri aşağıdaki gibi listeleyebiliriz:

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

Yığın 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 yığın uygulamasının nasıl yapılabileceği gösterilmiştir. Yığında tamsayı verileri saklayacak diziyeye *stack* adı verilmiş, dizinin boyutu 10 olarak tanımlan-

mıştır. Yığın yapısı için gerekli diğer bir değişken olan *top*, program başında -1 değeri almıştır. Yığına ekleme yapmak için *push()* fonksiyonu, yığından çıkarma yapmak için *pop()* fonksiyonu, yığının en tepesindeki elemanı elde etmek için ise *peek()* fonksiyonu tanımlanmıştır.

Ekleme işleminde öncelikle yığının doluluğu (dizinin son elemanının indisi olan $N-1$ ile *top* değişkenini karşılaştırarak) kontrol edilir ve yığında yer varsa işleme devam edilir. Yığına ekleme işlemi yapılabilecek ise *top* değişkeninin değeri 1 arttırılır, a tamsayısı dizinin *top* değerindeki indisine kaydedilir.

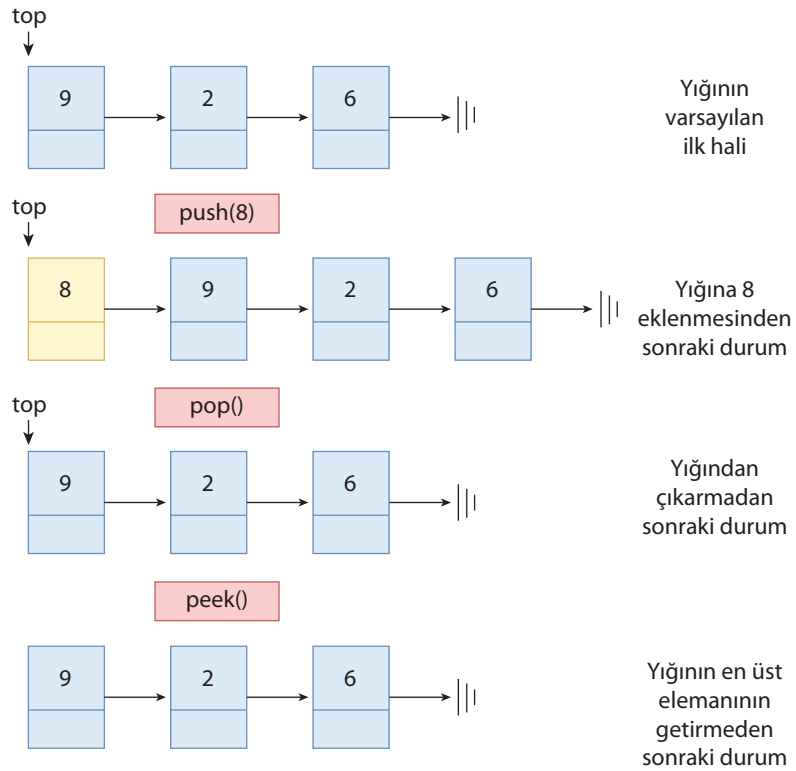
Çıkarma işleminde ve en üstteki elemanı elde ederken benzer işlemler yapılır. Öncelikle yığında eleman olup olmadığı kontrol edilir (*top* değişkeni ile 0 değerini karşılaştırarak). Yığında eleman varsa, a değişkenine dizinin *top* indisindeki değer atanır. Çıkarmada, *top* değişkeninin değeri 1 azaltılırken,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

Bağlı Liste ile Yığın Uygulaması

Yığın yapısını programlamak için bir bağlı listeden de faydalanılabilir. Bu yöntemde bağlı listenin elemanlarını oluşturacak bir veri yapısına ve yığının tepe noktasını takip edecek bir göstericiye ihtiyaç duyulur.

Şekil 2.10

Yığın Veri Yapısında Bağlı Liste Kullanımı.



Şekil 2.10'da yığın veri yapısının bağlı liste ile gösterimine yer verilmiştir. Yığının ilk halinde 9, 2 ve 6 sayılarını içeren elemanlar bulunmaktadır. Yığının tepe noktası, 9 sayısını içeren elemanı göstermektedir. Tanımlanan bu yığına, öncelikle *push* işlemi ile 8 sayısını içeren bir eleman eklenmiştir ve yığının 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 elemanı 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 yığın uygulamasının nasıl yapılabileceği gösterilmiştir. Yığında 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. Yığına ekleme yapmak için *push()* fonksiyonu, yığından çıkarma yapmak için *pop()* fonksiyonu, yığının 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 a tamsayısı, *t*'nin *data* değişkenine atanır. Bağlı liste boş ise (*top* göstericinin NULL olması), *top* göstericisi *t*'ye eşitlenir. Bağlı liste boş değilse, yığının en üzerinde *t* olacak şekilde tanımlama yapılır.

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

En üstteki elemanı elde ederken, yığının tepe noktası kontrol edilir. Yığında herhangi bir eleman varsa, yığının en üstünü işaret eden *top* göstericinin *data* değeri sonuç olarak döndürülür.

DİKKAT



Verilen örnekleri bilgisayar ortamında bizzat denemeniz, yığınlar hakkındaki deneyiminizin artmasına 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 eksiğine 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 gerekir. 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 başlangıcını işaret eden bir baş 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 ekleme (enqueue) ve kuyruktan eleman çıkarma (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 faydalanılabilir. Dizi kullanımı ile kuyruk programlarken, kuyruğun mevcut eleman sayısını takip etmek ve diziyi dairesel bir yapıya oturtmak gerekir. Bağlı liste kullanımı ile kuyruk programlamada ise kuyruğun eleman sayısının takibi gerekli değildir, ancak yazılan kod daha karmaşıktır.



Yığın yapısının mantığını kavramak ve temel yığın işlemlerini sıralamak

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

Yığın için tanımlı temel işlemler eleman ekleme (push), eleman çıkarma (pop) ve en üstteki elemanın değerini getirmedir (peek). Yığında ekleme ve çıkarma 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 Sıyalım

1. Aşağıdaki seçeneklerden hangisi 10 elemanlı bir dizinin ilk ve son indislerine ait numaraları göstermektedir?

- 0 ve 1
- 0 ve 9
- 0 ve 10
- 1 ve 9
- 1 ve 10

2. Aşağıdaki dizi tanımlamalarından hangisi derleyici tarafından derlendiğinde bir hata meydana gelir?

- double dizi[3];
- double dizi[3] = {};
- int dizi[] = {1, 2, 3};
- int dizi[4] = {1, 2, 3};
- int dizi[4] = {0, 1, 2, 3, 4};

3. Aşağıdakilerden hangisi diziler ve bağlı listelerin farklılık gösterdiği durumlardan **değildir**?

- Veri ekleme maliyeti
- Veri çıkarma maliyeti
- Hafıza kullanımı
- Değişken isimlendirme
- Verilere doğrudan erişim

4. Bir bağlı listede, listenin en sonundan en başına geçebilme imkanı varsa, o bağlı listenin hangi alt kategoriye ait olduğu söylenir?

- Tek yönlü bağlı liste
- Çift yönlü bağlı liste
- Dairesel bağlı liste
- Serbest bağlı liste
- 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?

- Listenin elemanlarını dolaşıp, ekrana yazmak
- Listeye yeni eleman eklemek
- Listenin ilk elemanını silmek
- Listenin son elemanını silmek
- head göstericisini sıfırlamak

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

- Dizi
- Tek yönlü bağlı liste
- Çift yönlü bağlı liste
- Yığın
- 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?

- C++
- Perl
- Ruby
- JavaScript
- Python

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

- Yığın
- Kuyruk
- Dizi
- Çift yönlü bağlı liste
- 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?

- Yığına yeni bir eleman eklemek
- Yığının en üzerindeki elemanı silmek
- Yığında bulunan elemanları ekrana yazdırmak
- Yığının tepe noktasını bir aşağı kaydırmak
- 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?

- 1
- 2
- 3
- 4
- 5

Kendimizi Sınavalım Yanıt Anahtarı

1. b Yanıtınız yanlış ise “Diziler” konusunu yeniden gözden geçiriniz.
2. e Yanıtınız yanlış ise “Diziler” konusunu yeniden gözden geçiriniz.
3. d Yanıtınız yanlış ise “Bağlı Listeler” konusunu yeniden gözden geçiriniz.
4. c Yanıtınız yanlış ise “Bağlı Listeler” konusunu yeniden gözden geçiriniz.
5. a Yanıtınız yanlış ise “Bağlı Listeler” konusunu yeniden gözden geçiriniz.
6. e Yanıtınız yanlış ise “Kuyruklar” konusunu yeniden gözden geçiriniz.
7. b Yanıtınız yanlış ise “Kuyruklar” konusunu yeniden gözden geçiriniz.
8. a Yanıtınız yanlış ise “Yığınlar” konusunu yeniden gözden geçiriniz.
9. e Yanıtınız yanlış ise “Yığınlar” konusunu yeniden gözden geçiriniz.
10. c Yanıtınız 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. sayi = %d\n", i+1, a[i]);
    }
}

int main(void) {
    printf("Lutfen %d adet tamsayi 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 gerekir.

```
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 yazdıran bir fonksiyon yazmaktır. 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 kuyruğa 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

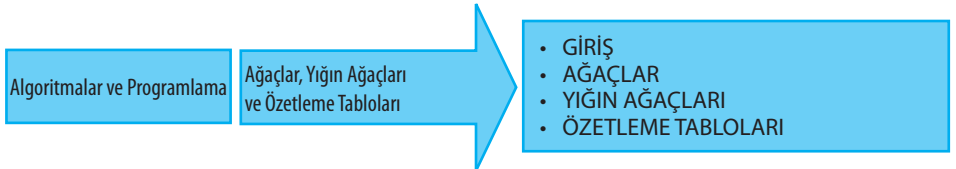
Bu üniteyi tamamladıktan sonra;

- 👁️ 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,
 - 👁️ Yığın ağaçlarının temel özelliklerini ve bu veri yapısında ekleme çıkarma işlemlerini özetleyebilecek,
 - 👁️ Özetleme tablosu veri yapısını ve özetleme fonksiyonlarını tanımlayabilecek,
 - 👁️ Özetleme fonksiyonlarında meydana gelen çatışmaları çözümleme yöntemlerini listeleyebilecek
- bilgi ve beceriler kazanabileceksiniz.

Anahtar Kavramlar

- İkili Ağaç
- İkili Arama Ağacı
- AVL Ağacı
- Yığın Ağacı
- Özetleme Tablosu
- Özetleme Fonksiyonu
- Çatışma
- Ayrık Zincirleme
- Açık Adresleme

İçindekiler



Ağaçlar, Yığın Ağaçları ve Özetleme Tabloları

GİRİŞ

Kitabımızın bu ünitesinde ağaçlar, yığın ağaçları ve özetleme 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 “Özetleme (Hash) Tabloları” bölümünde ise, verilerin anahtar ve veri çifti şeklinde saklanmasına imkan veren özetleme tabloları gösterilmiştir.

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

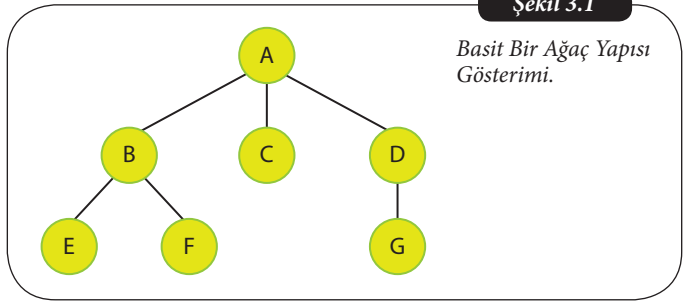
AĞAÇLAR

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 yolun uzunluğudur.
- Düğüm Yüksekliği (Height of a Node): Bir düğümden ağaçtaki en alt çocuğa kadar olan yolun uzunluğudur.



- **Düğüm Derinliği (Depth of a Node):** Bir düğümden ağaç köküne kadar olan yolun uzunluğudur.

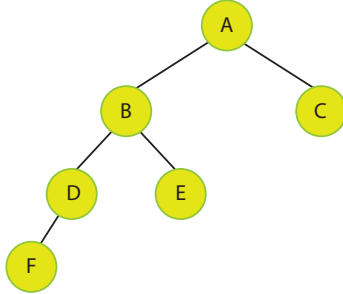
Şekil 3.1'de basit bir ağaç yapısı gösterilmiştir. Bu ağacın kökü A düğümüdür. B, C ve D düğümleri ise A'nın çocuklarıdır. E ve F düğümleri B'nin çocukları, G düğümü ise D'nin çocuğudur. C, E, F ve G düğümlerinin herhangi bir çocuğu bulunmamaktadır. Kök düğüm olan A'dan en alt çocuklara uzanan yolun uzunluğu 2 olduğu için, bu ağacın yüksekliği de 2'dir. F düğümünün 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 düğümün en fazla 2 çocuğa sahip olabildiği ağaç türüdür. Bu veri yapısında ekleme, silme ve arama işlemleri çok hızlı bir şekilde yapılabilir.

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

ÖRNEK 3.1

İkili ağaca 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 sayısını içeren bir ağaç düğümü oluşturulmuş ve bu düğüm kök olarak belirlenmiştir. Kökün sol çocuğuna 6 değerini içeren bir düğüm, sağ çocuğuna da 8 değerini 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 sıra 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



1

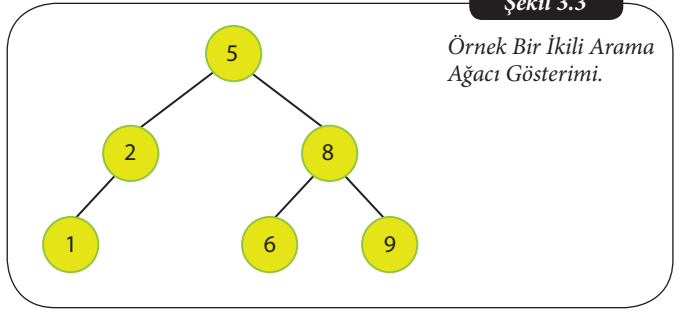
İkili ağaçlar üzerinde gezinme yöntemlerini pekiştirmek için Örnek 3.1'deki programa üç temel gezinme yöntemini ekleyiniz. Ana fonksiyon içerisinde 5 elemanlı bir ağaç oluşturup, oluşan ağaçtaki değerleri tüm gezinme yöntemlerini kullanarak ekrana yazdırınız.

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-küçüklük ilişkisi bulunmaktadır.

İkili ağaç özellikleri taşıyan bir ağacın ikili arama ağacı olabilmesi için ağaçtaki her düğümün, sol alt ağaçındaki tüm değerlerden büyük olması, sağ alt ağaçındaki 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 incelendiğinde, her bir düğümdeki değerin düğüme ait sol alt ağaçtaki değerlerden büyük, sağ alt ağaçtaki değerlerden küçük olduğu görülmektedir. İkili ağaç özelliklerini taşıyan ve içerdiği veriler arasında büyüklük-küçüklük kuralını sağlayan bu ağaç, ikili arama ağacıdır.



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

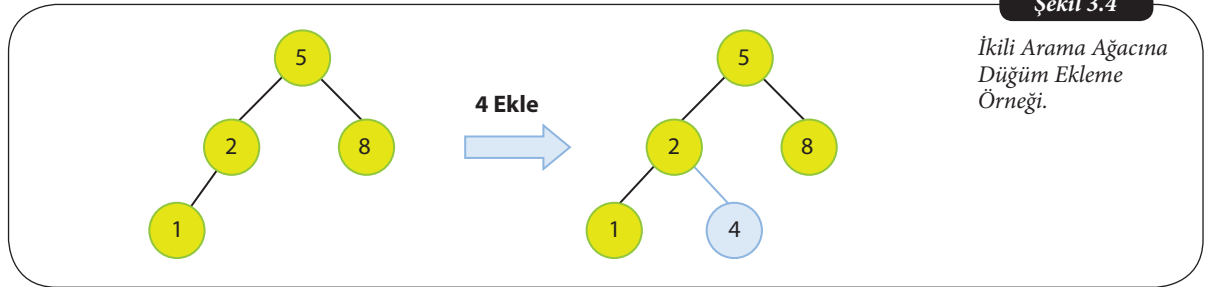


DİKKAT

İkili Arama Ağacına Düğüm Ekleme

İkili arama ağacına düğüm ekleme işlemi, ikili ağaca düğüm eklemekten daha karmaşıktır. İkili arama ağacına düğüm eklenirken, ikili arama ağacı veri yapısının özellikleri korunmalıdır.

İkili arama ağacına düğüm eklemede, ağacın düğümlerinin değerleri arasındaki büyüklük-küçüklük ilişkisini korumak için eklenecek düğümün yeri tespit edilmelidir. Ağaç boş ise yeni düğüm, ağacın kökü olarak tayin edilir. Ağaç dolu ise ağacın kökünden yola çıkılarak, eklenecek düğümün değeri ile kök düğümün değeri karşılaştırılır. Yeni düğümün değeri kökteki değerden büyükse sağ alt ağaca, küçükse sol alt ağaca doğru ilerlenir. Bu işlem, yeni düğüm için uygun bir yer bulunana kadar tekrarlanır.



Şekil 3.4'te bir ikili arama ağacına yeni bir düğüm ekleme örneği gösterilmiştir. Bu örnekte var olan ağaca, 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 ağaca gidilir. Sol alt ağacın kökünde yer alan düğümün değeri 2 olup bu değer 4'ten küçüktür. Yeni düğümü eklemek için sağ alt ağaca gidilir, sağ alt ağaç boş olduğu için yeni düğüm buraya eklenir. İkili arama ağacını temsil edecek veri yapısı ve ağaca 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

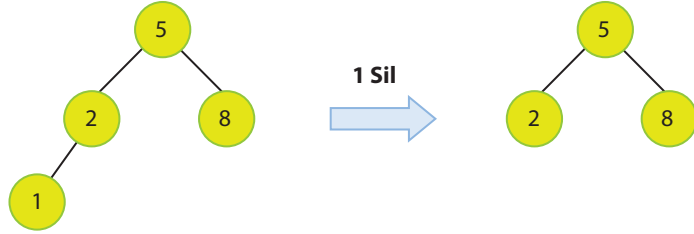
İkili arama ağacından düğüm çıkarma için öncelikle düğümün ağaçta bulunması gerekir. Düğüm ağaçta yer alıyorsa, ikili arama ağacı özellikleri korunarak çıkarma işlemi gerçekleştirilir.

İkili arama ağacından düğüm çıkarırken incelenmesi gereken üç durum vardır:

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

Şekil 3.5

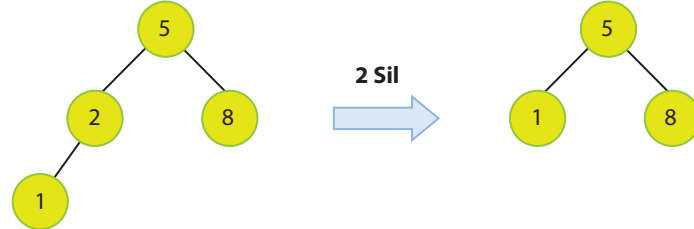
İkili Arama Ağacından Çocuğu Olmayan Düğümü Çıkarma Örneği.



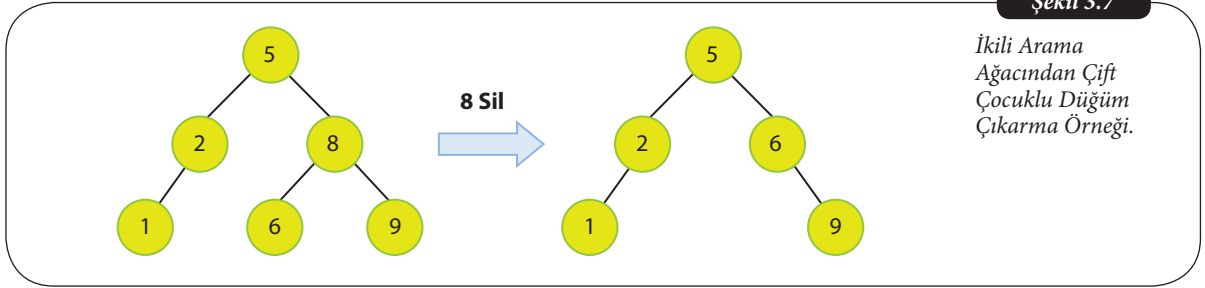
- Çıkarılacak düğümün 1 çocuğu var ise; düğümün çocuğundan itibaren olan alt ağaç düğümün ebeveynine bağlanır, düğüm hafızadan silinir.

Şekil 3.6

İkili Arama Ağacından Tek Çocuklu Düğüm 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 çıkarılabilmektedir.

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 düğümüne bağlanmıştır.

Şekil 3.7'de gösterilen örnek, ikili arama ağacından düğüm çıkarma işlemlerinin en karmaşık olanıdır. 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 düğümün 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 geliştirme yapabilirsiniz. Programınızdaki ağaca düğüm ekleme, ağaçtan düğüm çıkarma ve ağacı ekrana yazdırma fonksiyonlarını deneyip, sonuçlarını gözlemleyiniz.



AVL Ağaçları

İkili ağaçlarda ve ikili arama ağaçlarında ağacın yüksekliği için herhangi bir ölçüt bulunmamaktadır. N adet düğümüne sahip bir ikili ağacın 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.

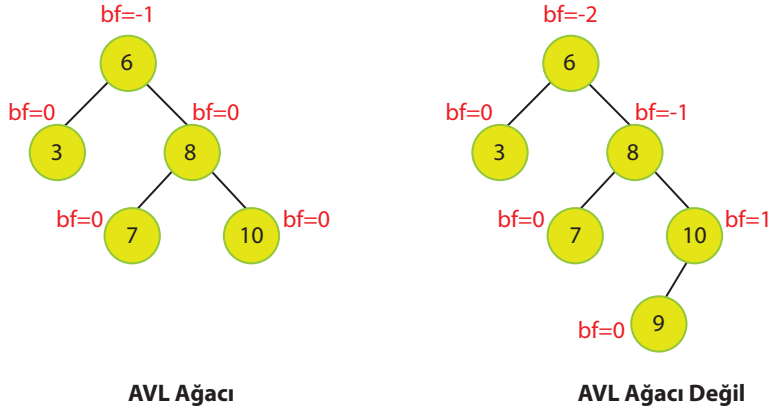
AVL ağaçlarındaki düğümler için **denge faktörü** aşağıdaki formül ile hesaplanır ve dengeli bir ağaç için bu değerler yalnızca -1, 0 ve 1 olabilir:

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

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

Şekil 3.8

AVL Ağacı Olan ve AVL Ağacı Olmayan Ağaç Örnekleri.



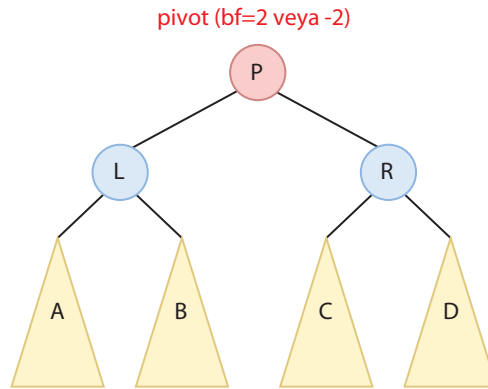
Şekil 3.8'de iki adet ikili arama ağacı gösterilmiştir. Her iki ağaç için de denge faktörleri hesaplanmış, ilgili düğümün üzerinde kırmızı ile belirtilmiştir. Şekilde verilen ilk ağaçtaki tüm denge faktörleri 0 ve -1 değerlerinden oluşmaktadır. Dolayısıyla ilk ağaç, bir AVL ağacıdır. Şekildeki ikinci 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.

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

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

Şekil 3.9

AVL Ağacında Pivot ve Sol-Sağ Alt Ağaçlar.



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 faydalanılacaktır:

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

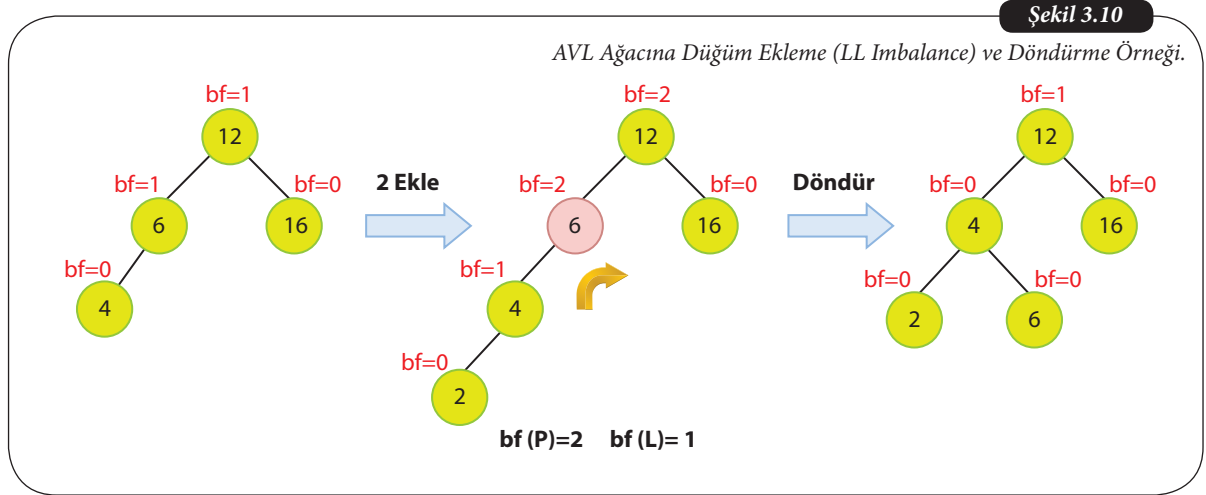
- A bölgesi: Pivotun sol alt ağacının sol çocuğudur.
- B bölgesi: Pivotun sol alt ağacının sağ çocuğudur.
- C bölgesi: Pivotun sağ alt ağacının sol çocuğudur.
- D bölgesi: Pivotun sağ alt ağacının sağ çocuğudur.

AVL Ağacına Düğüm Ekleme ve Denge Korunumu

AVL ağacına düğüm eklerken, öncelikle düğümün ekleneceği yer bulunur. Düğüm eklendikten 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:

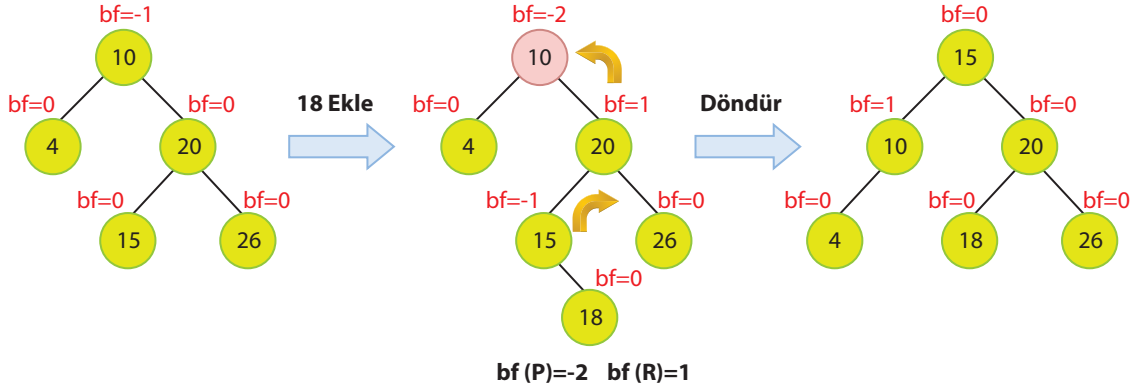
- A bölgesine ekleme (LL Imbalance): P'nin denge faktörü 2, L'nin denge faktörü 0 veya 1 iken karşılaşılr, pivot etrafında sağa doğru tek döndürme ile çözülür.
- D bölgesine ekleme (RR Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 0 veya -1 iken karşılaşılr, pivot etrafında sola doğru tek döndürme ile çözülür.
- C bölgesine ekleme (RL Imbalance): P'nin denge faktörü -2, R'nin denge faktörü 1 iken karşılaşılr, sağ ve sol çift döndürme ile çözülür.
- B bölgesine ekleme (LR Imbalance): P'nin denge faktörü 2, L'nin denge faktörü -1 iken karşılaşılr, 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 için ağaçta LL Imbalance vardır. Dengesizliğin çözümü için pivot etrafında sağa doğru tek döndürme işlemi uygulanır. Elde edilen son ağaç dengededir ve AVL ağacı veri yapısı özelliklerini taşımaktadır.

Şekil 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

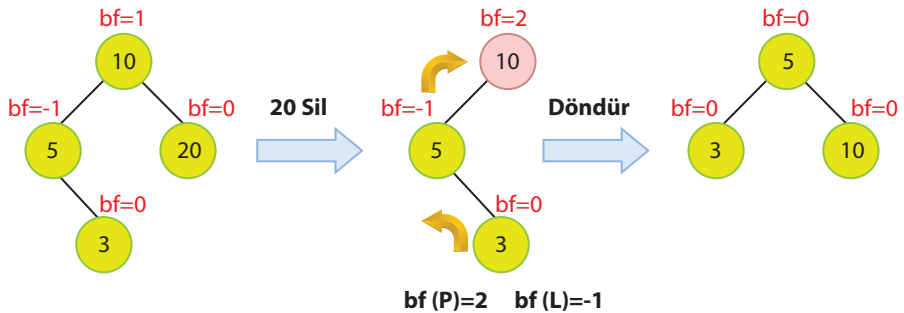
AVL ağacından düğüm çıkarılırken, ikili arama ağacındaki çıkarma yöntemi izlenir. Düğüm çıkarıldıktan sonra oluşan yeni ağaç üzerinde denge faktörleri hesaplanır. Ağaçta bir dengesizlik olması durumunda dengesizliğin olduğu düğüm (pivot) tarafında bir veya iki tane döndürme işlemi uygulanır.

Çıkarma işleminden sonra oluşan dengesizlik, dört ayrı şekilde görülebilir:

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

Şekil 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 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, L'nin denge faktörü -1 olduğu için ağaçta LR Imbalance vardır. Dengesizliğin çözümü için önce 3 değerini içeren düğüm etrafında sola döndürme, sonrasında pivot etrafında sağa döndürme işlemi uygulanır. Elde edilen son ağaç dengededir ve AVL ağacı veri yapısı özelliklerini taşımaktadır.

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



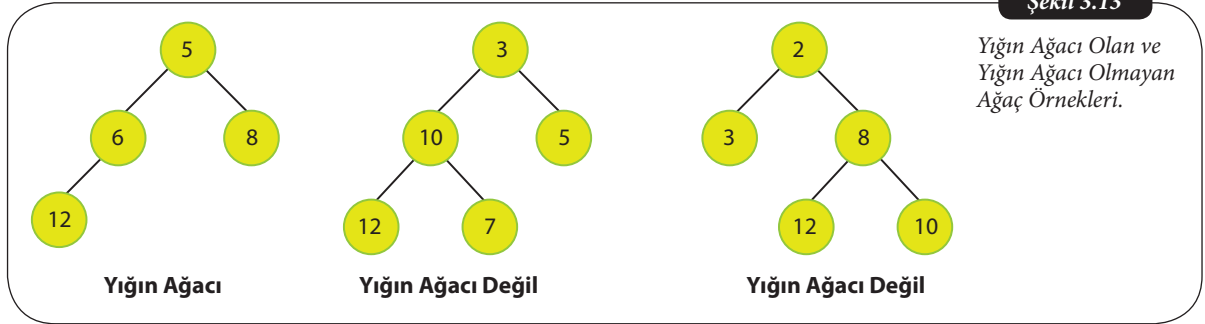
DİKKAT

YIĞIN AĞAÇLARI

Yığın 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 ağaca eleman ekleme işlemleri hızlıca yapılabilir.

Aşağıda verilen iki özelliği sağlayan bir ikili ağaç, yığın ağacı veri yapısı olarak sınıflandırılır:

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



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 yığın ağacıdır.
- Şekilde verilen ikinci ağaçta ağaç bütünlüğü sağlanmasına rağmen, heap özelliği sağlanmamaktadır. 10 değerine sahip düğümün 7 değerine sahip çocuğu olduğu için, bu ağaç bir yığın ağacı değildir.
- Şekilde gösterilen üçüncü ağaçta son düzeydeki düğümlerin soldan sağa doğru dolu olmadığı görülmektedir. Dolayısıyla bu ağaç için bir bütünlük söz konusu değildir. Ağaç bütünlüğü sağlanmadığı için, bu ağaç bir yığın ağacı değildir.

Bu konuda anlatılan yığın ağaçları, minimum yığın ağaçlarını kapsamaktadır. Yığın ağaçlarında en büyük elemanı baz alarak da işlem yapılabilir. Ağacın en üst noktasında en büyük elemanın yer aldığı, heap özelliğinde ise büyük olma durumunun arandığı yığın ağaçları maksimum yığın ağaçları olarak adlandırılır.



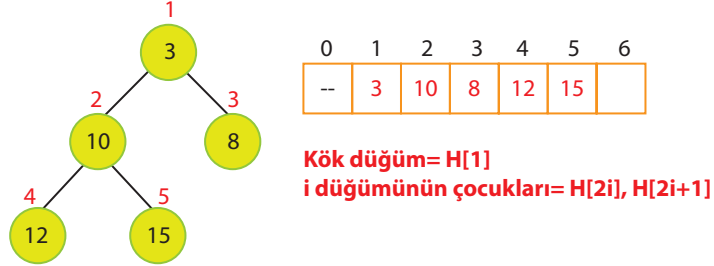
DİKKAT

Dizi ile Yığın Ağacı Uygulaması

Yığın ağaçları, ağaç bütünlüğüne sahip ikili ağaçlar olduğu için, yığın 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 yığın ağacının bir dizi ile ifade ediliş biçimi gösterilmiştir. N uzunluğunda bir dizinin 1 numaralı indisine yığın ağacının kökü yerleştirilir. Ağacın geri kalan 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

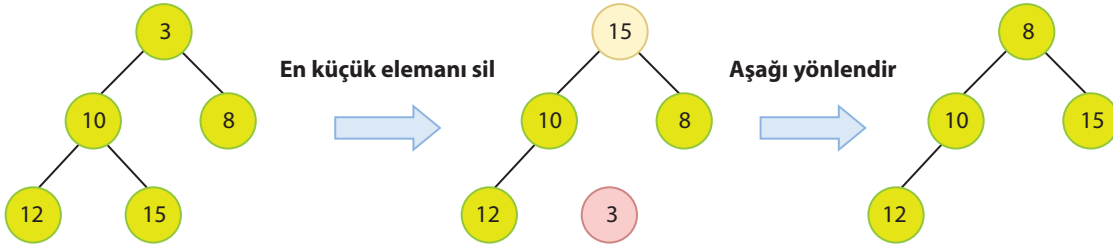
Yığın ağacının en küçük elemanı, ağacın kökünde yer almaktadır. Bu elemanı elde etmek için dizinin 1 numaralı indisine erişmek yeterlidir. N uzunluğunda H isimli bir dizi ile ifade edilen yığın ağacında en küçük elemanı elde etmek için dizinin H[1] elemanına erişmek yeterlidir.

Yığın Ağacından En Küçük Elemanı Çıkarma: Aşağı Yönlendirme

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

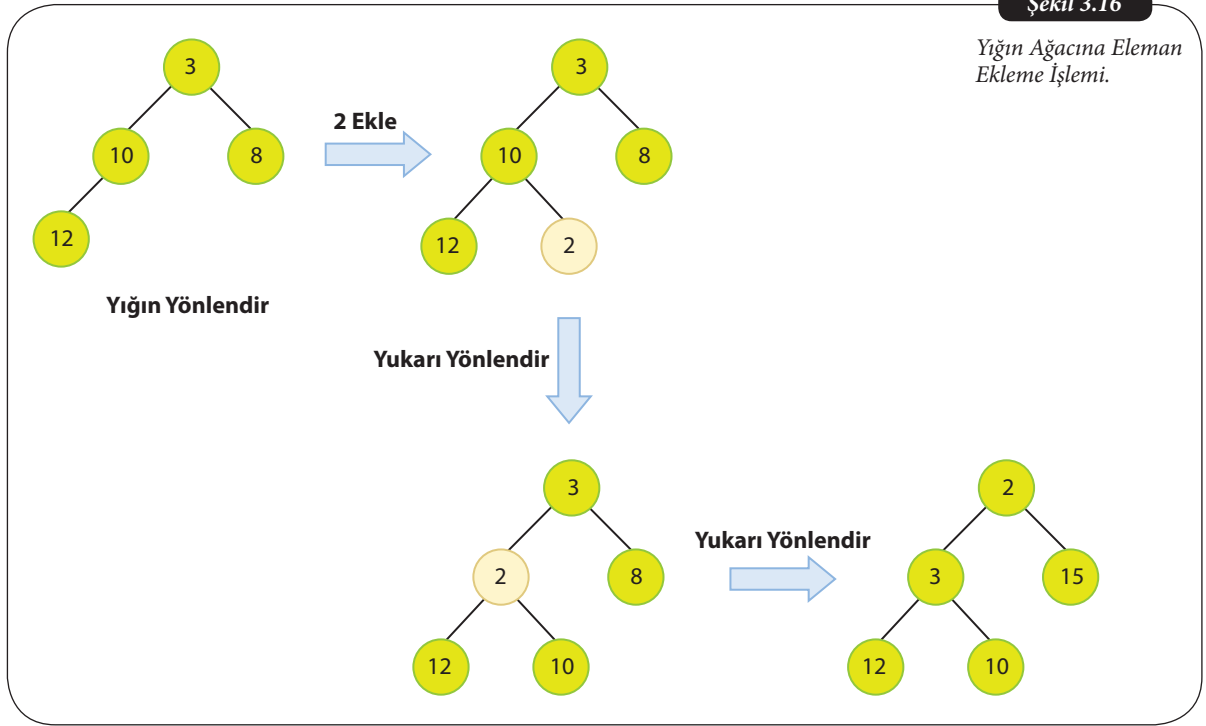
Şekil 3.15

Yığın Ağacından En Küçük Elemanı Çıkarma İşlemi.



Yiğın Ağacına Eleman Ekleme: Yukarı Yönlendirme

Yiğın ağacına yeni bir eleman eklerken, ağacın son düzeyinin en sağında yeni bir düğüm yaratılır ve eklenecek değer bu yeni düğüme atanır. Oluşan yeni ağaç için ağaç bütünlüğü korunmakta, fakat ağacın heap özelliği kaybolmaktadır. Ağacın heap özelliğini sağlaması amacıyla, eklenen düğüm üzerinden yukarı yönlendirme işlemi yapılır. Eklenen yeni düğüm, ağaç içerisinde doğru yere gelinceye kadar ebeveyniyle karşılaştırılır ve düğümün değeri ebeveynin değerinden küçük ise düğüm ile ebeveynin yeri değiştirilir. Şekil 3.16'da, yiğın ağacına eleman ekleme işleminin gösterildiği bir örnek verilmiştir.



ÖZETLEME (HASH) TABLoları

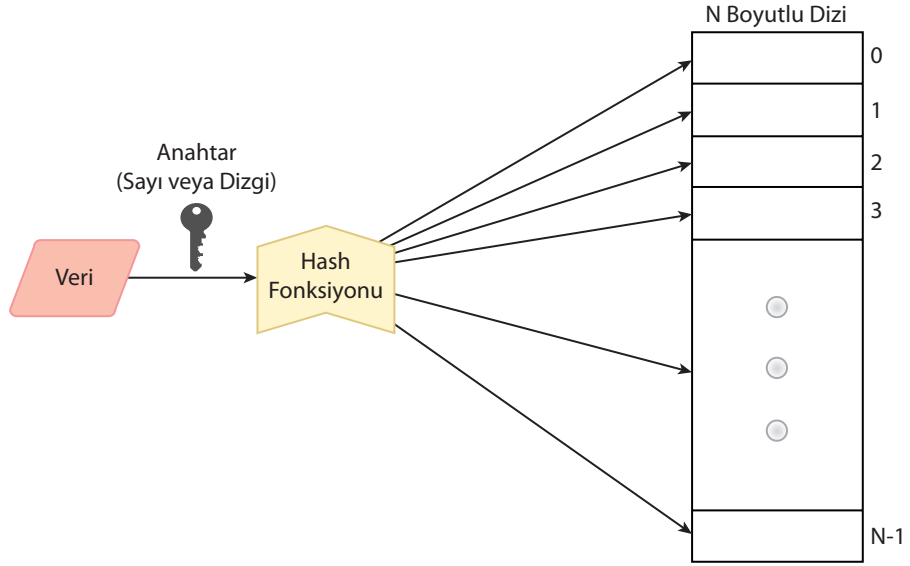
Özetleme tabloları ekleme, silme ve arama işlemlerinin çok hızlı bir şekilde yapılmasını sağlayan, verileri bir anahtar ve veri çifti şeklinde saklayan veri yapısıdır.

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

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

Şekil 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 sayısının 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, dizideki 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 algoritmaya sahiptir. Fonksiyon için tanımlı algoritma, fonksiyona verilen anahtar değeri doğrultusunda bir indis değeri 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 çatışma senaryosu gözlemlemek için Örnek 3.6'da verilen hash fonksiyonunu inceleyelim. Özetleme 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, çatışma meydana gelmiştir.

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

- Fonksiyon içerisinde hesaplamalar hızlı yapılmalıdır.
- Hesaplama sonucu üretilen değerlerde minimum çatışma olmalıdır.
- Özetleme tablosundaki tüm alanlar kullanılabilir olmalıdır.
- Özetleme tablosundaki doluluklarda eşit dağılım sağlanmalıdır.

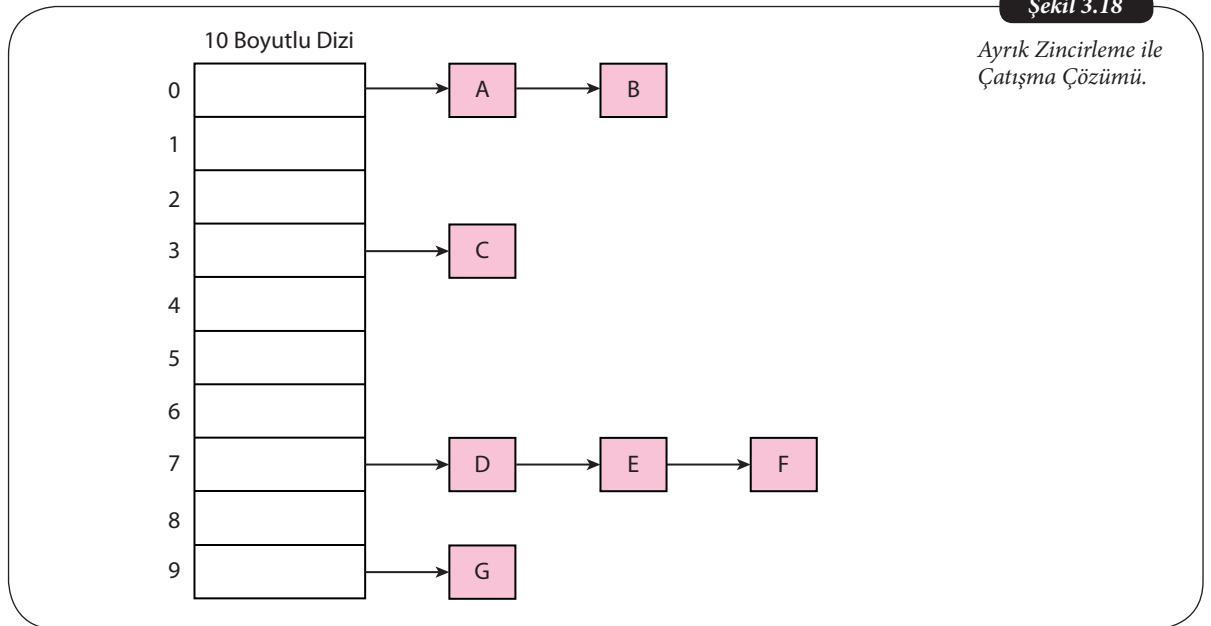
Çatışma Çözüm Yöntemleri

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

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

Ayrık Zincirleme (Separate Chaining)

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



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

- hash(anahtar A) = hash(anahtar B) = 0
- hash(anahtar C) = 3
- hash(anahtar D) = hash(anahtar E) = hash(anahtar C) = 7
- hash(anahtar G) = 9

Listelenen varsayımlar doğrultusunda yaşanan çatışmaların çözümü ayrık 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 çatışma 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 kaçınıcı kez gönderildiğini ifade eden sayıdır. F fonksiyonu çatışma çözümü için kullanılan ikinci fonksiyondur ve $F(0) = 0$ 'dır.

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

- Doğrusal Sınama (Linear Probing): $F(i)=i$
- Karesel Sınama (Quadratic Probing): $F(i)=i^2$
- İkili Hash (Double Hashing): $F(i)=i*Hash_2(i)$

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 denemede ($i=0$) sırasıyla 8, 9 ve 0 numaraları indislere eklenmiştir. 29 anahtarına sahip veri, ilk denemede ($i=0$) 9 numaralı indise, ikinci denemede ($i=1$) 0 numaralı indise eklenmeye çalışılmış fakat tablonun bu alanları dolu olduğu için çatışmalar meydana gelmiştir. 29 anahtarına sahip veri üçüncü denemede ($i=2$) 1 numaralı indise eklenmiştir. 30 anahtarına sahip veri ilk denemede ($i=0$) 0 numaralı indise, ikinci denemede ($i=1$) 1 numaralı indise eklenmeye çalışılmış fakat tablonun bu alanları dolu olduğu için çatışmalar meydana gelmiştir. 30 anahtarına sahip veri üçüncü denemede ($i=2$) 2 numaralı indise eklenmiştir.

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 bağlandığı hiyerarşik bir veri modelidir. Düzgünlerden 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 yapılabilir.

İkili arama ağaçları, ikili ağaçların özel bir türüdür. Bu veri yapısında, ikili ağaç özelliklerine ek olarak düğümlerde yer alan veriler arasında büyüklük-küçüklük ilişkisi bulunmaktadır. İkili ağaç özellikleri taşıyan bir ağacın ikili arama ağacı 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 ağacın düğümlerini belirli bir algoritma ve sıra çerçevesinde dolaşma anlamına gelir. İkili ağaçlarda gezinme yapılırken özyinelemeli fonksiyonlardan faydalanılır 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 ağacı 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 içerisindeki denge korunmakta, sol alt ağaç ile sağ alt ağaç arasındaki yükseklik farkı en fazla 1 olabilmektedir.

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

AVL ağacı üzerinde yapılan düğüm ekleme ve çıkarma işlemleri ağacın dengesini bozabilir. Ağacın 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 karşılaşılr, 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 karşılaşılr, 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 karşılaşılr, sağ ve sol çift döndürme ile çözülür.
- B bölgesinde ekleme/çıkarma (LR Imbalance): P'nin denge faktörü 2, L'nin denge faktörü -1 iken karşılaşılr, 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 ağacın yığın ağacı olarak sınıflandırılabilmesi için aşağıda verilen iki özelliği sağlaması gerekir:

- Ağaç bütünlüğü: Ağacın son düzeyi hariç tüm düzeyleri, içerdikleri düğümler bakımından eksiksiz olmalıdır. Ağacın 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 bütünlüğüne sahip ikili ağaçlar olan yığın ağaçları, bir dizi kullanarak programlanabilir. Ağacın kökünde yer alan ağacın en küçük elemanını elde etmek için dizinin 1 numaralı indisine erişmek yeterlidir.

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

Yığın ağacına yeni bir eleman eklerken, ağacın son düzeyinin en sağında yeni bir düğüm yaratılır ve ekle-

necek deęer bu yeni düęüme atanır. Ağacın heap özelliğini sağlamak için eklenen düęüm 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 yapılır.

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



Özetleme fonksiyonlarında meydana gelen çatışmaları çözümleme yöntemlerini listelemek

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

Bir hash fonksiyonu için çatışma oluşumu ihtimalini ortadan kaldırmak mümkün olmasa da ayırık 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 sınıma, karesel sınıma veya ikili hash olabilir.

Kendimizi Sınavalım

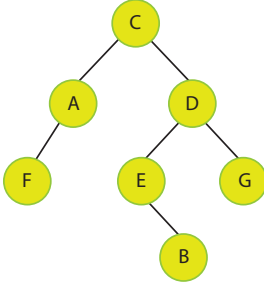
1. Ağaç veri yapısında yer alan düğümleri birbirine bağlayan kenarlara ne ad verilir?

- Çocuk
- Ebeveyn
- Dal
- Kök
- Ağaç yüksekliği

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

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

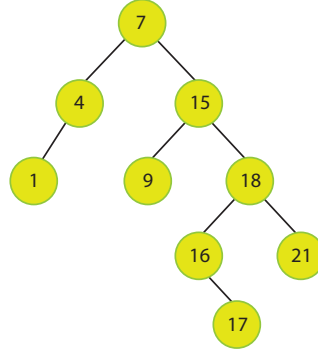
3.



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

- CAFDEBG
- CADFEGB
- FACBEDG
- FABEGDC
- ABCDEF G

4.



Yukarıdaki şekilde gösterilen ikili arama ağacının yüksekliği kaçtır?

- 1
- 2
- 3
- 4
- 5

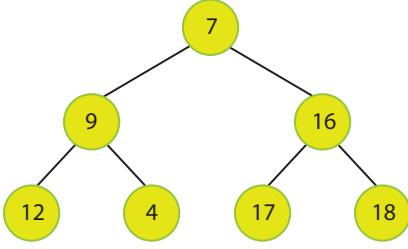
5. AVL ağaçları ile ilgili aşağıda verilen ifadelerden hangisi doğru **değildir**?

- AVL ağaçları, ikili arama ağaçlarının özel bir türüdür.
- Bir AVL ağacının kökünde yer alan düğümü silmek mümkündür.
- Bir AVL ağacı her zaman dengeli olmalıdır.
- Ağaca eklenen her düğüm sonrası bir döndürme yapılır.
- Ağaç düğümlerinin denge faktörü -1, 0 veya 1 olabilir.

6. Aşağıdaki ifadelerden hangisi yığın ağaçları için geçerlidir?

- Her ikili arama ağacı, aynı zamanda bir yığın ağacıdır.
- Yığın ağacı veri yapısı oluşturmada bir diziden faydalanılamaz.
- Ağaca eleman ekledikten sonra aşağı yönlendirme işlemi yapılır.
- Ağacın en küçük elemanını sildikten sonra yukarı yönlendirme işlemi yapılır.
- Yığın ağacında sağ çocuğu olup, sol çocuğu olmayan bir düğüm olamaz.

7.



Yukarıda gösterilen ağaç üzerinde aşağıdaki işlemlerden hangisi uygulandığında, yığın 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
- Bağlı liste
- Özetleme tablosu
- İkili arama ağacı
- Kuyruk

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

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

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

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

Kendimizi Sınavalım Yanıt Anahtarı

1. c Yanıtınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
2. e Yanıtınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
3. a Yanıtınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
4. d Yanıtınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
5. d Yanıtınız yanlış ise “Ağaçlar” konusunu yeniden gözden geçiriniz.
6. e Yanıtınız yanlış ise “Yığın Ağaçları” konusunu yeniden gözden geçiriniz.
7. b Yanıtınız yanlış ise “Yığın Ağaçları” konusunu yeniden gözden geçiriniz.
8. c Yanıtınız yanlış ise “Özetleme Tabloları” konusunu yeniden gözden geçiriniz.
9. a Yanıtınız yanlış ise “Özetleme Tabloları” konusunu yeniden gözden geçiriniz.
10. e Yanıtınız yanlış ise “Özetleme 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 ağaca düğüm eklenebilir ve ağaçtan düğüm silinebilir. Programa eklenecek bir gezinme yöntemi ile ağacın son halindeki düğümler ekrana 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 dugum bulunamadi.\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 sınıma 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

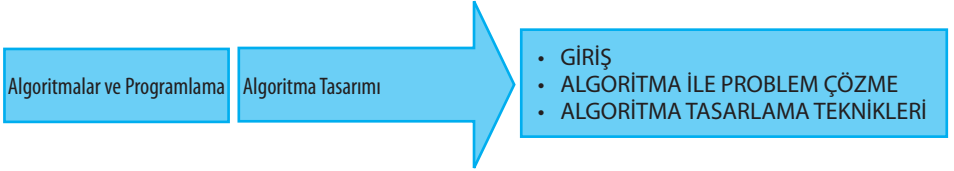
Bu üniteyi tamamladıktan sonra;

- 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



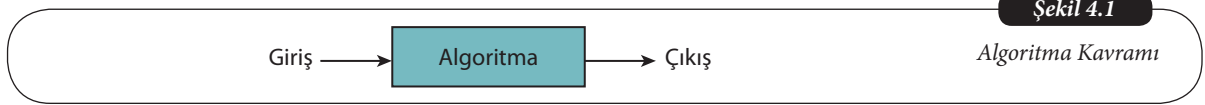
Algoritma Tasarımı

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 tasarlanacak 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.



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 varsayalım. Hangi yolları kullanırsak en kısa mesafede istediğimiz yere ulaşabiliriz? Günümüzde artık yol tarifleri için navigasyon 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 arttırmak için çeşitli şifreleme algoritmaları kullanmaktadırlar.

Akıllı telefonların kullanımını yaygınlaştıkça 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 odaklama yapılabilmektedir.

Teknolojik Olarak Algoritmalar

Algoritmalarla çalışırken kendi kendimize şu soruyu sorabiliriz: Bilgisayarlarımız sonsuz hıza sahip ve bilgisayar hafızaları ücretsiz olsaydı yine de algoritmalar üzerinde çalışmamıza gerek olur muydu? Bu soruya vereceğimiz cevap şüphesiz evet olurdu. Bulduğumuz çözüm yönteminin doğru cevap ile sonuçlandığını gösterebilmek için de algoritmalarla çalışmamız gerekmektedir. Her ne kadar günümüzde bilgisayarlar oldukça hızlı olsa da sonsuz hızda değiller 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 olmayacaktır. 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 tasarımı 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 kolaylaştırabilecektir. Karşılaşılabilecek bütün özel durumları da düşünmemiz gerekmektedir.

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

Problemi anlamak, algoritma tasarımının birinci aşamasıdır.

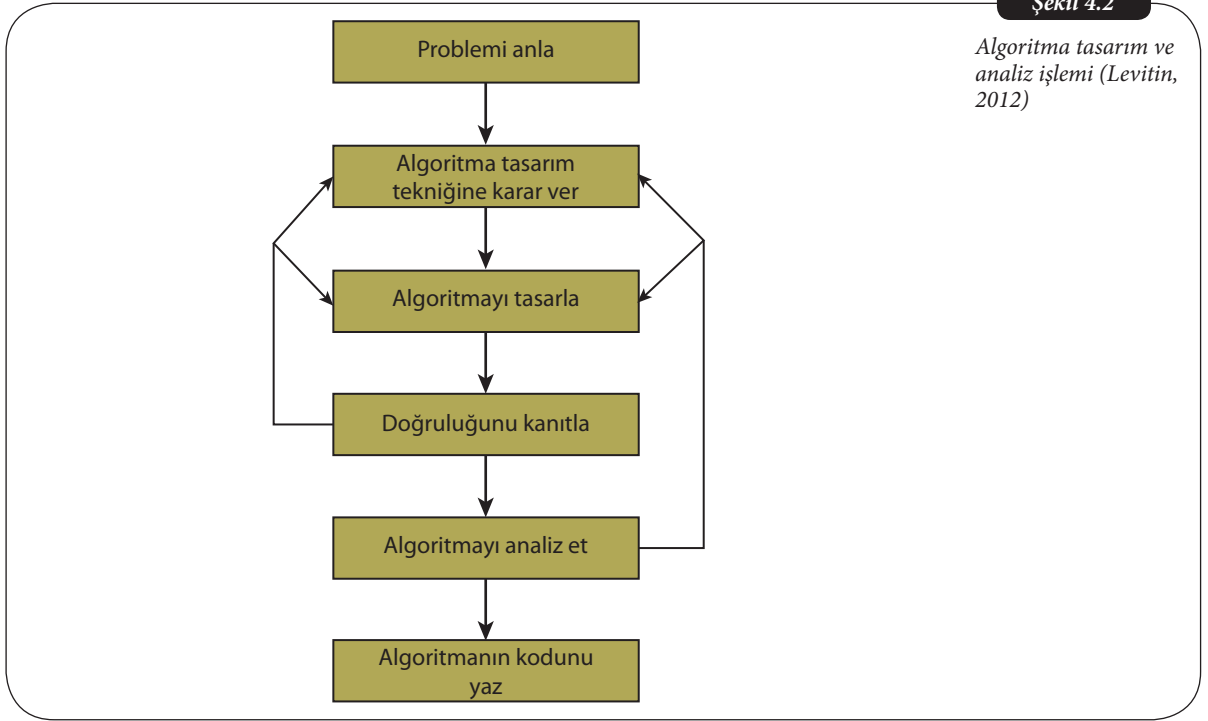
SIRA SİZDE



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

Şekil 4.2

Algoritma tasarım ve analiz işlemi (Levitin, 2012)



Problemi bütün ayrıntılarıyla anladıktan sonra düşünmemiz gereken husus, tasarlayacağımız algoritmayı hangi konfigürasyona sahip bilgisayarda çalıştırmak istediğimizdir. Kullanacağımız bilgisayarın işlemci hızı, hafıza kapasitesi vb. özellikleri tasarlayacağımız algorithmada etkilidir. Problemimiz çok karmaşık ve çözümünü 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.

Kodlayacağımız algoritmayı tasarlarken, kodu çalıştıracığımız bilgisayarın konfigürasyonu çok önemlidir.

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

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) $ortalama = (sayı1 + sayı2 + sayı3) / 3$
- 3) Kullanıcıya *ortalama* değerini göster

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



2

SIRA SİZDE

Tasarlanan algoritmanın doğru şekilde çalıştığı mutlaka teyit edilmelidir.

Algoritma tasarımı ve analizi sürecinin son adımı ilgili algoritmanın kodunu yazmaktır.

Tasarladığımız algoritmayı göstermenin diğer bir yöntemi de akış diyagramı kullanmaktır. 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, tasarımı bu doğrultuda yapmamız gerekmektedir. Ünitimizin 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 gerekir. 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 tasarımda göz önünde bulundurmalıyız. Örneğin, bölme işlemi yapan algoritmanın, sıfıra 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 faydalanmamız mümkün olabilir. Hazır fonksiyonlar beklentilerimizi tam olarak karşılamıyor ise kendi fonksiyonlarımızı kodlamamız gerekecektir. Kodu yazdıktan sonra belirli testlerden geçirmek, programın güvenilirliğini arttıracaktır.

ALGORİTMA TASARLAMA TEKNİKLERİ

Algoritmaları temel olarak iki gruba ayırabiliriz. 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 özellikleri 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 yapabiliriz. Başlangıçta, dizinin ilk elemanını en büyük kabul ederiz. Daha sonra, en büyük kabul ettiğimiz değeri sırasıyla dizinin bütün elemanlarıyla karşılaştırırız. Herhangi bir sıradaki elemanın değeri, en büyük kabul ettiğimiz değerden büyükse en büyük eleman değerimizi güncelleriz. Dizinin sonraki elemanlarını karşılaştırırken 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.

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 yapmaktayız. Daha sonra bir döngü içerisinde dizinin bütün elemanlarını sırasıyla bu toplam değerine ekliyoruz. Dizinin sonuna geldiğimizde elimizdeki toplam değeri, aradığımız sonucu oluşturmaktadır.



SIRA SİZDE

```
#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 düşünelim 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 benzetilmektedir. Şekil 4.3'te bu algoritmanın çalışmasına bir örnek verilmiştir. Bu algoritmada, problemi birer birer küçülterek çözüme ulaşıyoruz. Örnekten 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 koyarız. Böylece dizideki eleman sayısı kadar iterasyon geçtiğinde, dizimiz sıralı hâle gelir.

Şekil 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]);
    }
}
```

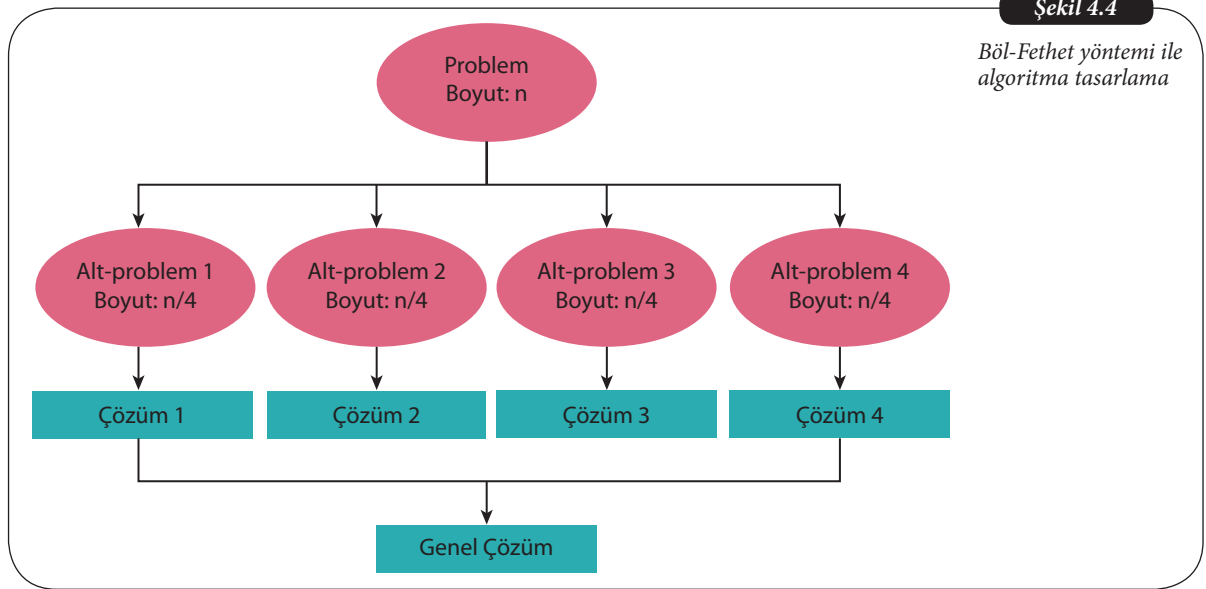
Küçült-Fethet algoritma tiplerinin farklı türleri vardır. Araya sokma sıralama algoritmasında problemin çözümüne birer birer yaklaşıldı. Her adımda bir tane sayı, o anki sıralı yerine koyuldu. Farklı uygulamalar için problemin çözümüne, farklı veya değişken sabitlerle azaltma yaparak da ulaşılabilir.

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 oluştuğunu varsayalım. 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.



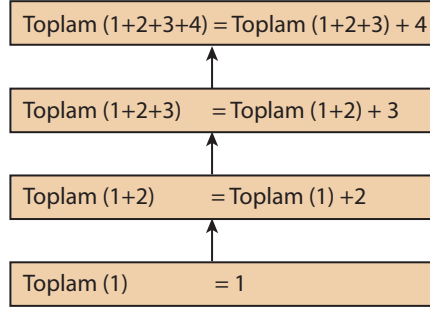
Böl-Fethet yönteminde öncelikli olarak 1'den N 'ye kadar olan sayıların toplamını hesaplayan bir program yazalım. Bu problemi özyinelemeli fonksiyon olarak yazalım:

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

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 sayısını 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 hesapladıktan sonra geriye doğru gelerek bulduğumuz sonuçları birleştirerek ana sonuca ulaşmaktayız.

Şekil 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;
}

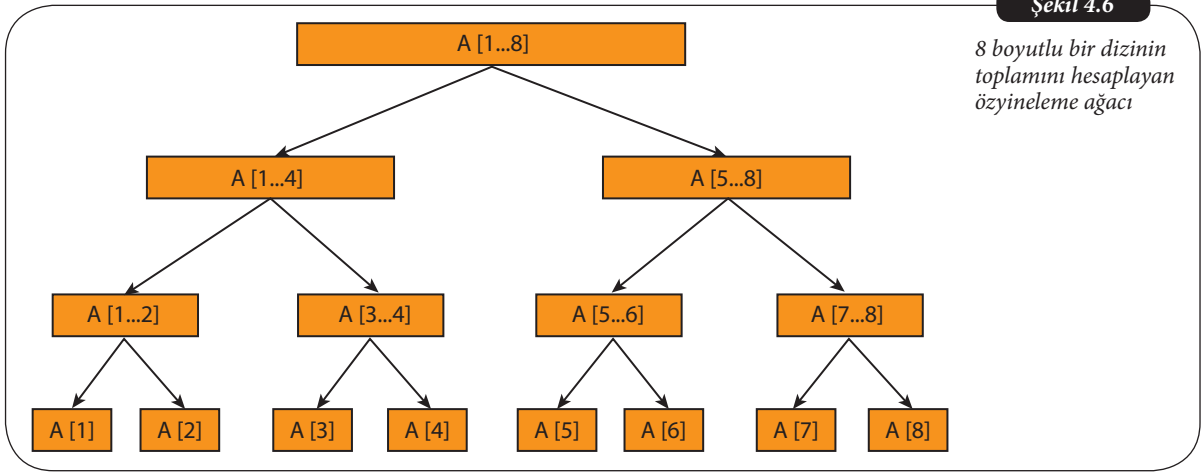
void main()
{
    int x = 0;

    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 ulaşabiliriz. Şekil 4.6'da dizideki eleman sayısı 8 olduğundaki özyineleme ağacının yapısı görünmektedir. 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);

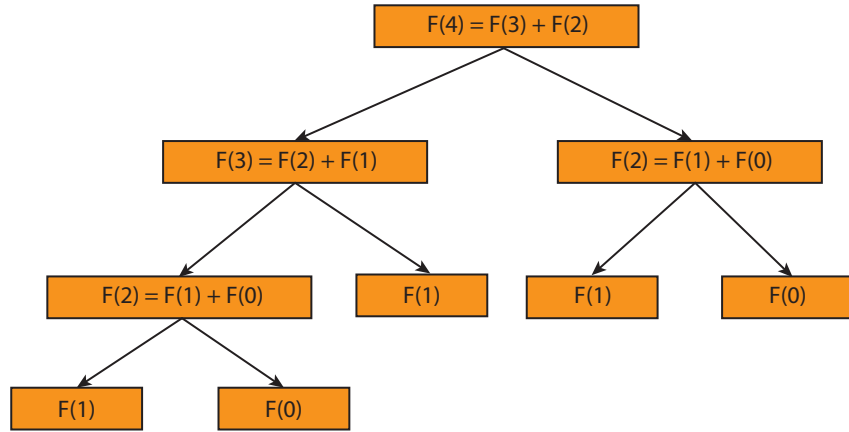
    printf("Dizinin elemanlarının toplamı : %d\n", a);
}
```

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));
    }
}
  
```

Özet



Algoritmaların hesaplamadaki rolünü açıklamak

Kitabımızın bu ünitesinde öncelikle algoritmaların hesaplamadaki ve problem çözmedeki rolü anlatıldı. Matematiksel olarak çözebileceğimiz 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 tekniğine karar vermek, algoritmayı tasarlamak, doğruluğunu kanıtlamak, algoritmayı analiz etmek ve uygulamayı geliştirmektir. Ünitimizde 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 aşağıdan yukarıya ya da yukarıdan aşağıya doğru olabileceği ifade edildi. Bu teknikte, problemin küçük parçalarını adım adım çözerek sonuca ulaşmaktayız. Bu tekniğe 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 bulunduğundan 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 tasarladıktan sonra, tasarlanmış olduğumuz algoritmanın doğruluğunu teyit etmemiz, algoritmayı çalışma zamanı ve hafıza gereksinimi açısından analiz etmemiz gerekmektedir.

Kendimizi Sınavalım

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 çizdiğimizde, $F(4)$ 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 fonksiyonu 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 tasarımı ve analiz aşamalarının ilki aşağıdakilerden hangisidir?

- Problemi anlamak
- Tekniğe karar vermek
- Uygulamayı gerçekleştirmek
- Doğruluğunu kanıtlamak
- Algoritma analizi yapmak

6. Algoritma tasarımı ve analizi sürecinde, algoritmayı tasarladıktan sonraki aşama aşağıdakilerden hangisidir?

- Problemi anlama
- Tekniğe karar verme
- Uygulamayı gerçekleştirme
- 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

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;
}
```

- Dizideki tüm sayıların toplamını
- Dizideki negatif sayıların toplamını
- Dizideki pozitif sayıların toplamını
- Dizideki eleman sayısını
- Dizideki en büyük elemanı

Kendimizi Sınyalım Yanıt Anahtarı

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 Tasarlama” konusunu yeniden gözden geçiriniz.
4. e Yanıtınız yanlış ise “Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet Yöntemi ile Algoritma Tasarlama” 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 Tasarlama” konusunu yeniden gözden geçiriniz.
9. b Yanıtınız yanlış ise “Özyinelemeli Fonksiyon Algoritmaları ve Böl-Fethet Yöntemi ile Algoritma Tasarlama” 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 karşılaşılabileceğimiz özel durum, dizinin kapasitesinin dolu olmasıdır. Örneğin, dizinin kapasitesi 100 iken diziye 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. $\text{delta} = b^2 - 4ac$
3. $\text{delta} < 0$ ise reel kök yoktur yaz ve 6. adıma git
4. $x_1 = \frac{-b - \sqrt{\text{delta}}}{2a}$, $x_2 = \frac{-b + \sqrt{\text{delta}}}{2a}$
5. x_1 ve x_2 değerlerini ekrana yaz
6. Dur

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;
}
```

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;
}
```

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.

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

- GİRİŞ
- ALGORİTMA ANALİZİ İÇİN MATEMATİKSEL ARKA PLAN
- FONKSİYONLARIN BÜYÜMESİ
- ALGORİTMALARIN EN KÖTÜ DURUM, EN İYİ DURUM VE ORTALAMA DURUM VERİMLİLİKLERİ
- ASİMPOTİK GÖSTERİMLER
- ÖZYİNELEMELİ OLMAYAN ALGORİTMALARIN ANALİZİ
- ÖZYİNELEMELİ ALGORİTMALARIN ANALİZİ

Algoritma Analizi

GİRİŞ

Kitabımızın önceki ünitesinde farklı algoritma tasarım yöntemleri anlatılmıştı. Bu ünite-mizde ise algoritma analizi konusu ele alınacaktır. Bir problem için algoritma tasarlarken bu programı bilgisayarda çalıştırdığımızda, problemin çözümüne ulaşılmasının ne kadar zaman alacağı önemli bir husustur. Bilindiği üzere, bir problemin farklı algoritmalarla birden fazla çözümü olabilir. Bir problemi çözmek istediğimizde, olası algoritmalarla 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 çalıştıracaksak, programcının en kolay tasarlayacağı algoritmayı seçmesi en basit yaklaşımdır. Ancak sıralama probleminde milyonlarca veri olması söz konusuysa 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 ulaşabileceği, kullandığımız bilgisayarın konfigürasyonuna ve kullandığımız derleyicinin ürettiği makine koduna bağlı olarak değişebilmektedir. Çalışma zamanı karşılaştırması için kullanılacak 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 ünite-mizde, algoritmadaki temel operasyonun analizini yapacağız. Ünite-mizde 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 ulaşılabilmeye 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 hesaplamalarında 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 sıklıkla kullanılan fonksiyonların girdi değerlerine karşılık gelen çıktıları verilmiştir. Fonksiyona bağlı olarak, girdi değeri büyüdükçe sonucun ne kadar farklılaştığı 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 alacaktır. 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	3.3×10	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 değişebilmektedir. Örneğin, bir lineer sıralama algoritmasını ele alalım. Bu algoritma, verilen bir sayının dizideki elemanlardan biri olup olmadığının sonucunu döndürmektedir. Bu algoritmada verilen sayı, dizideki ilk elemandan başlayarak sırasıyla dizinin tüm elemanlarıyla karşılaştırılmaktadır. Bulmak istediğimiz değer dizideki ilk elemana eşit olması, arama probleminde karşılaşılabileceğimiz en iyi durumdur. Böyle bir durumda, dizinin ilk elemanı ile karşılaştırmayla hemen sonuca ulaşip dizinin diğer elemanlarıyla karşılaştırmaya gerek kalmayacaktır. Bu durum, lineer arama algoritmasının optimal girdi için verimliliğidir. 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 düşünecek 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 hesaparken 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 çalınma sürelerinin ortalamasını alırız. Böylece, ortalama durum verimliliğini elde etmiş oluruz. Lineer arama algoritmasını düşünecek olursak verilen girdiye göre 1., 2., 3., ..., N. gibi adımlarda aradığımız sayıyı bulabiliriz. Bu olasılıkların hepsinin ortalamasını aldığımızda (N/2)'nci adımda, yani dizinin orta elemanında sonucu bulmamız gerekmektedir. Bu değer ise lineer arama algoritmasının ortalama durum verimliliğidir.

SIRA SİZDE



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

ASİMPOTİK GÖSTERİMLER

Bir algoritma çalıştığında kaç birim adımda sonuca ulaşacağı hesaplanarak aynı problemi çözen farklı algoritmaların verimliliği karşılaştırılabilir. Bilgisayar bilimcileri, fonksiyonların büyümesini 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 adımına 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.

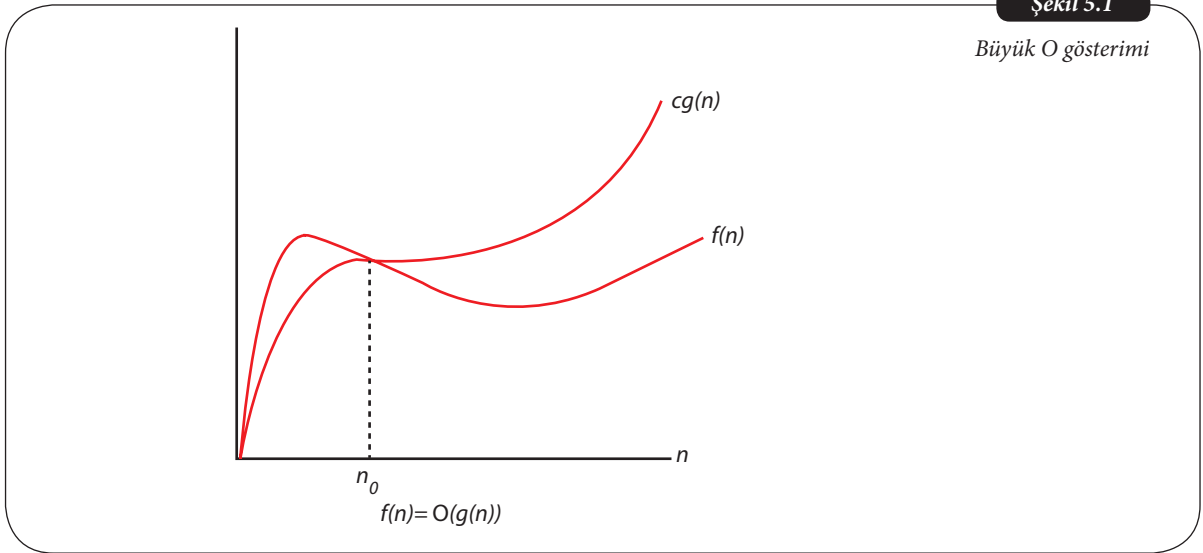
$$O(g(n)) = \{f(n): 0 \leq f(n) \leq cg(n), n \geq n_0\}$$

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

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

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

Görüldüğü üzere $f(n) = n + 6$ fonksiyonu $O(n)$ 'dir.



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?

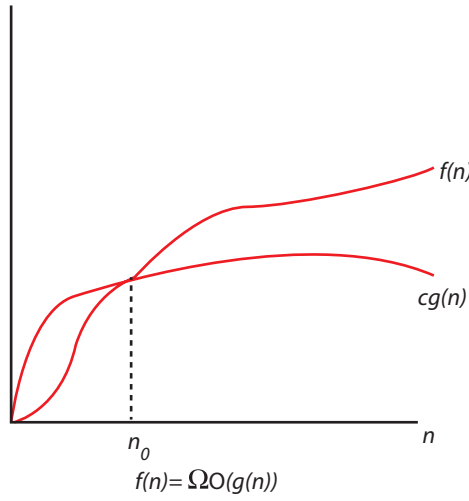
$f(n)$ fonksiyonunun $\Omega(n^2)$ olduğunu gösterebilmek için aşağıdaki eşitsizliği sağlayacak n_0 ve c pozitif katsayılarını bulmamız gerekmektedir.

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

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 5.2Büyük Ω Gösterimi**Büyük Θ Gösterimi**

Büyük Θ gösterim, daha önce incelenmiş olan O ve Ω gösterimlerinin birleşimine benzetilmektedir. 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, Θ 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.

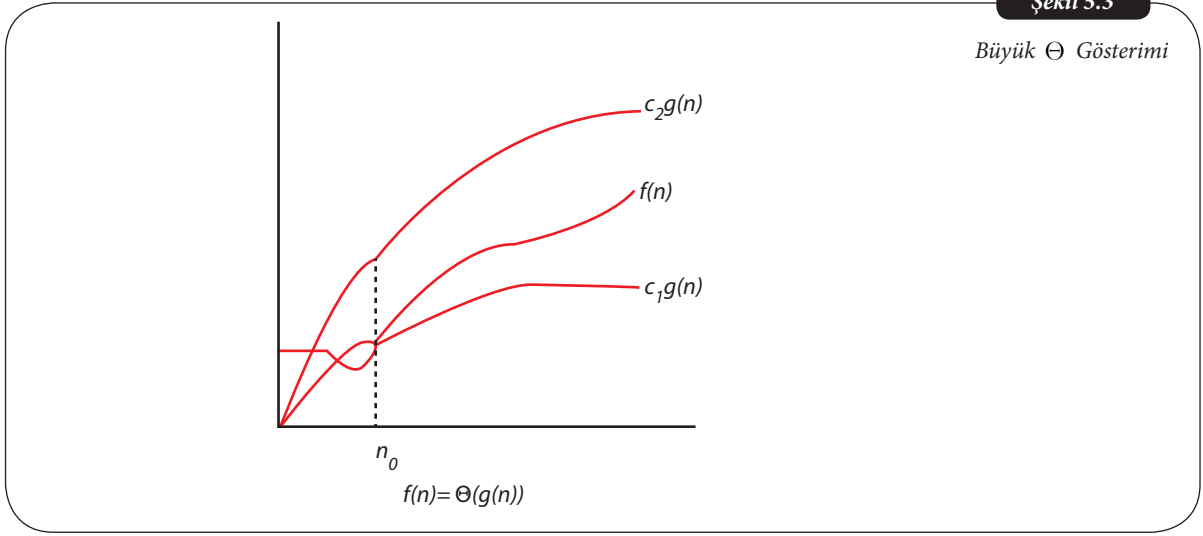
$$\Theta(g(n)) = \{f(n): c_1 g(n) \leq f(n) \leq c_2 g(n), n \geq n_0\}$$

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 çıkabilmektedir. Logaritmik sınıf algoritmalarında ise problemin büyüklüğü her bir iterasyonda belirli bir sayıya (genellikle 2) bölünmektedir. Lineer sınıftaki algoritma, dizide verilen bütün elemanların üzerinden tek tek gitmektedir. Yarı doğrusal sınıftaki algoritmalar çoğunlukla böl-birleştir algoritmalarıdır. İkinci derece (kareli) sınıftaki algoritmalarda iki tane iç içe döngü bulunmaktadır. Üçüncü derece (kübik) sınıftaki algoritmalarda üç tane iç içe döngü bulunmaktadır. Üstel sınıftaki algoritmalar, bir kümenin bütün alt kümeleri üzerinde işlem yapmaktadırlar. Faktöriyel algoritmalarında bir kümenin bütün permütasyonları 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

4

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 $O(n)$ 'dir.

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 $O(n^2)$ 'dir.

İki tane arka arkaya for döngüsü: Arka arkaya iki tane döngü söz konusu olduğunda, iki döngünün de çalışma sayıları bulunur ve bu sayılar birbiriyle 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 olmaktadır. 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 $O(1)$ 'dir. if/else deyiminin bir bütün olarak zaman karmaşıklığı $O(n)$ olur.

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



SIRA SİZDE

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

Diğer örneğimiz lineer arama üzerinedir. Bu algoritma, daha önce açıkladığımız algoritmalarından biraz farklılık göstermektedir. Temel operasyon *for* döngüsünün bulunduğu kısımdır. Ancak *for* döngüsünün içerisinde arama yaparken sonuca ulaşıldığında döngüden çıkmaktadır. Yani döngü en kötü durumda, n defa dönmektedir. En iyi durumda ise ilk iterasyonda sonuca ulaşmaktadır. Ancak algoritma analizi yaparken en kötü duruma dikkate alınmaktadır. Böylece, algoritmanın bu değerden daha uzun süre çalışmayacağı garanti edilmektedir. En kötü duruma dikkate alınarak 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:

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

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 tasarlanacağı 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 değişmeyeceği belirlenir.
4. Başlangıç koşulları ile birlikte algoritmanın özyinelemeli fonksiyon bağıntısı yazılır.
5. Fonksiyonların büyümesi ve toplam ifadeleri kullanılarak özyineleme bağıntısı çözümler 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 $O(N)$ mertebesindedir.

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 $O(n)$ mertebesindedir.

Özyinelemeli algoritmalar için verilebilecek diğer bir örnek, ikili arama algoritmasıdır. Bu algortmada 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 karşılaştırmaktayız. Aradığımız eleman ortadaki elemana eşit ise ortadaki elemanı cevap olarak döndürülür. 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 bölme 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 yapılırken, hangi zaman biriminin dikkate alınacağı önemli bir konudur. Algoritmaları karşılaştırırken süreyi saniye, dakika vb. zaman birimi kullanarak hesaplırsak, aynı algoritmayı farklı bir bilgisayarda çalıştırdığımızda 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 durumdaki 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 ölçümü yapılabilir. Olası tüm girdiler için analiz yapıp ortalaması alındığında ise ortalama durum verimliliği söz konusudur.



Asimptotik 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 ulaşmayacağını garanti etmektedir. Büyük Ω gösterimi ise algoritmanın çalışma zamanının alt sınırını ifade etmektedir. Büyük Ω 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 büyüklüğünü veren parametre belirlenir. Daha sonra algoritmanın temel operasyonu tespit edilir. Bu temel operasyonun sadece girdi parametresine bağlı olup olmadığının ve başka değerlere göre değişip değişmediğinin analizi yapılır. Bu durumlar dikkate alınarak algoritma için işlem adımlarının sayısı bulunur. Bu değer genellikle toplam ifadeyle gösterilir. Toplam ifadeleri için verilen standart formüller ve kurallar kullanılarak algoritmanın zaman karmaşıklık ifadesi elde edilir.

Kendimizi Sıyalım

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

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

Sıra Sizde 2

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
2^{10}	10	2^{10}	$10 \cdot 2^{10}$	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.

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

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

Görüldüğü 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

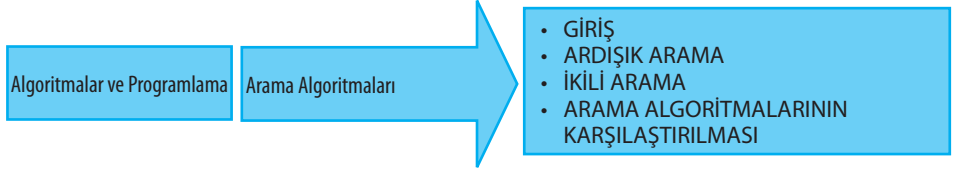
Bu üniteyi tamamladıktan sonra;

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

Anahtar Kavramlar

- Arama
- Ardışık Arama
- İkili Arama

İçindekiler



Arama Algoritmaları

GİRİŞ

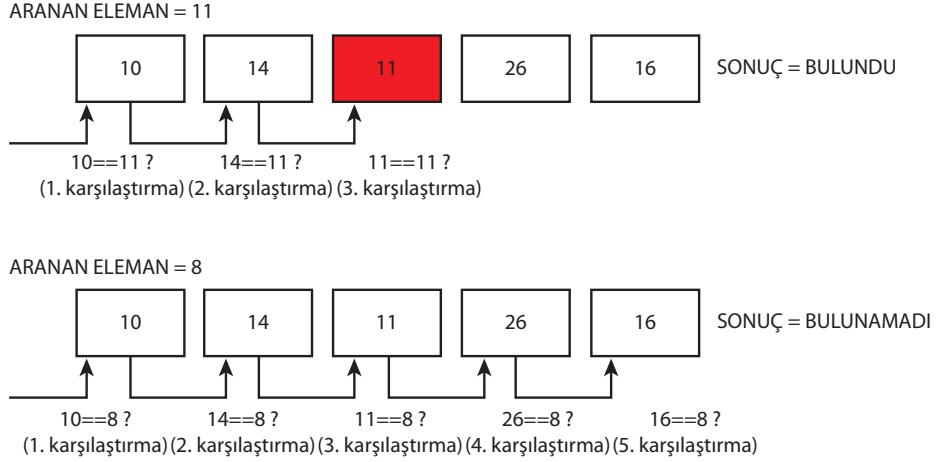
Arama, genel olarak dizilerin veya herhangi bir veri yapısının içerisinde bir elemanın bulunup bulunmadığının tespiti şeklinde ifade edilebilir. Söz konusu elemanın veri yapısı içerisinde bulunduğu tespit edilirse konum bilgisinin elde edilmesi de söz konusu olabilir. Arama algoritmaları, arama işlemini 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 gerektiği bilinmektedir. Sıralı durumda olmayan bir diziyeye bu türdeki bir arama algoritmasının uygulanması için dizinin öncelikle sıralı hale getirilmesi gerekir. 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 karşılaştırılması ilkesine dayanır. Dizinin sonuna ulaşılmadan aranan elemanın bulunması söz konusu olursa algoritma sonlandırılır. Aranan elemanın bulunamadığı durumda ise bu kontroller dizinin başından sonuna kadar sürdürülecektir. 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 konumundaki eleman ile karşılaştırılır. Aranan elemanın değerinin dizinin elemanlarından birisiyle aynı olduğu görülürse algoritma başarılı bir şekilde sonlandırılır. Eğer aranan eleman dizinin içerisinde mevcut değilse, karşılaştırmalar dizinin ilk elemanından son elemanına kadar sürecektir. 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. elemanı 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 edildiği 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("%dsayisi, 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 sayisi, dizinin 3. konumundadir.
8 sayisi dizide bulunamadi.

```

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

- *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 gerekir. İkili arama algoritması çalışmaya başladığında, ilk olarak dizinin ilk ve son elemanlarının konumları tespit edilir. Bu bilgilerden faydalanılarak orta elemanın konumu hesaplanır. Daha sonra, aranan eleman dizinin orta elemanı ile 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. Orta elemanın konumunun tam sayı olması zorunludur. Bu örnek hesaplamada, iki sayının ortalaması olan 5.5 sayısının ondalıklı kısmı bu sebeple atılmakta ve bir tam sayı elde edilmektedir. Algoritmanın

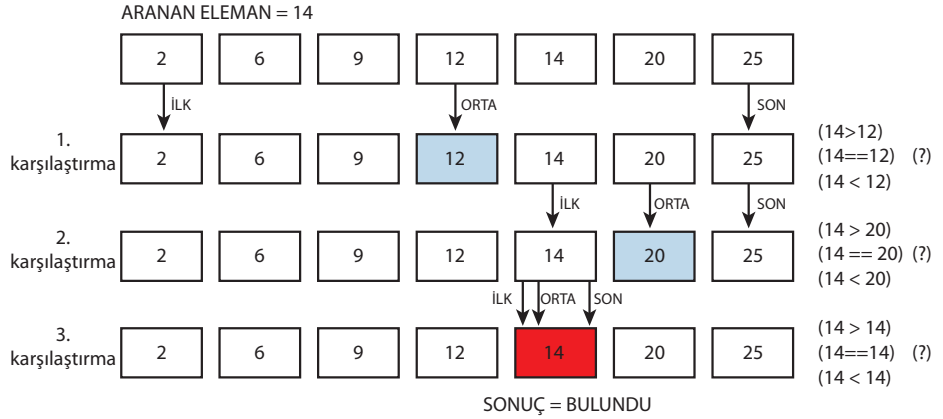


SIRA SİZDE

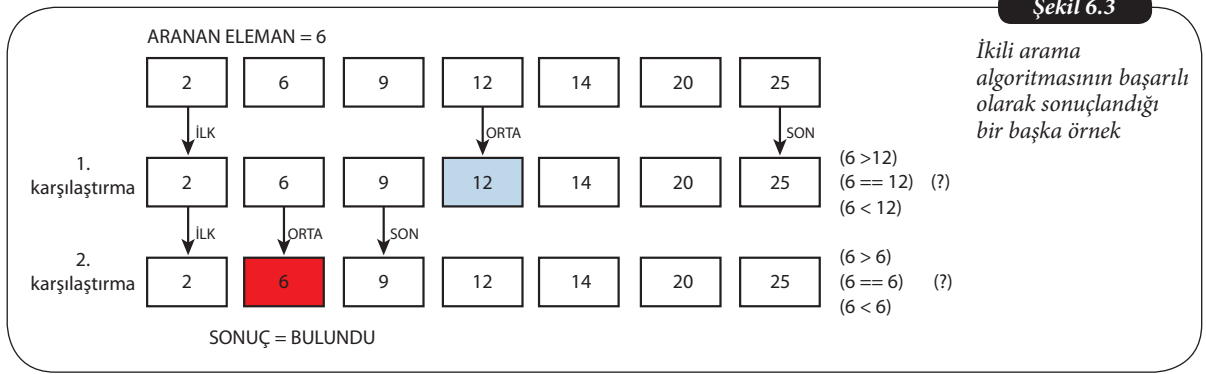
çalışması tamamlandığında aranan eleman dizide bulunamamışsa, arama işlemi başarısız bir şekilde sonlandırılır. Şekil 6.2'de ikili arama algoritmasının başarılı sonuçlandığı bir örnek verilmiştir. Bu örnek, dizinin küçükten büyüğe doğru sıralı olduğu durum için gerçekleştirilmiştir. Dizi, büyükten küçüğe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

Şekil 6.2

İkili arama algoritmasının başarılı olarak sonuçlandığı bir örnek

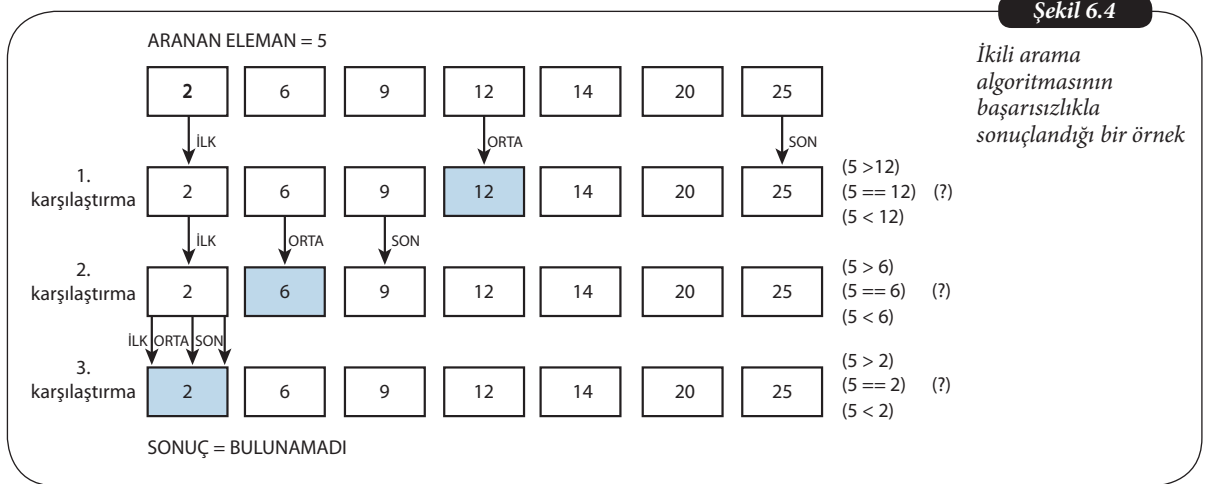


İkili arama algoritmasının çalışma mantığının anlatıldığı bu örnekte 7 elemanlı bir dizi bulunmaktadır. Söz konusu dizi içerisinde 14 sayısının aranması söz konusudur. Algoritma çalışmaya başladığında ilk, orta ve son elemanların konumları belirlenmiştir. Şekilden de görüleceği üzere dizinin 4. konumundaki 12 sayısı orta elemandır. Daha sonra, aranan eleman olan 14 sayısının orta elemene eşit, küçük veya büyük olması durumuna bakılmaktadır. Bu sayı, orta elemandan büyük olduğu için aramaya dizinin ortası ile son 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 anlaşıldığından aramaya dizinin ilk ve orta elemanı arasındaki elemanlardan devam edilir. Bu noktada da son elemanın bilgisi, bir önceki orta eleman olan 20 sayısından 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 sonlandırılacaktır. 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 sonlandırılacaktır. 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 sayılardan 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 anlaşıldığından 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 şekillerde açıklandığı üzere, 7 elemanlı bir dizinin oluşturulduğu ve 3 farklı sayının bu dizi içerisinde *ikili_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 içerdiği 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 sayısı, dizinin 5. konumunda

İkinci arama işlemi:

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

Üçüncü 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_özyinelemeli.c* program kodları çalıştırıldığında önceki ekran görüntüsünün aynı ortaya çıkmaktadır.

Özyinelemeli (recursive)

fonksiyon: Özyinelemeli fonksiyonlar, kendi içlerinde tekrar kendilerini çağıran 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_özyinelemeli.c */
#include <stdio.h>

/* İkili arama fonksiyonu */
void ikili_arama_özyinelemeli(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 görülmekteydi. 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 çıktığına kontrol ediniz.

- `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.



SIRA SİZDE

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 durumdaki 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 durumdaki 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 varsayalım. 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 değerlendirildiğinde ikili arama algoritması tercih edilmelidir.

Zaman karmaşıklığı (time complexity): Algoritmaların sonuca ulaşması için gerekli olan zaman hakkında bilgi veren bir ölçüttür. En kötü durumdaki 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.

- Program kodundaki `main` fonksiyonu içerisindeki `dizi` değişkeninin içerdiği 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.



SIRA SİZDE

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 tespitidir. Arama algoritmaları ise arama işlemini farklı yollarla gerçekleştiren algoritmalarlardır. Bu üniteye sayısal veriler içeren diziler üzerinde arama yapmaya yarayan algoritmalarından bahsedilmiştir.



Temel arama algoritmalarını uygulamak

Bu üniteye, 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 elemanı ile 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 elemanının konumu hesaplanır. Daha sonra, aranan eleman dizinin orta elemanı ile 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 bölümü arasında veya orta ile son bölümü 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 çalışabilmeleri mümkündür. Bu bölümde sadece en kötü durumdaki zaman karmaşıklıkları ele alınmıştır. Ardışık aramanın 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 aramaya 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 Sıyalım

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?
 - a. 3
 - b. 4
 - c. 5
 - d. 6
 - e. 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?
 - a. 6
 - b. 8
 - c. 10
 - d. 12
 - e. 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?
 - a. Yalnızca I
 - b. Yalnızca III
 - c. I ve II
 - d. II ve III
 - e. 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?
 - a. 2
 - b. 4
 - c. 6
 - d. 8
 - e. 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?
 - a. 1
 - b. 4
 - c. 6
 - d. 7
 - e. 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?
 - a. Yalnızca I
 - b. Yalnızca III
 - c. I ve II
 - d. II ve III
 - e. 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 yapılırken 2. karşılaştırma adımında ilk, orta ve son elemanların konum bilgileri ne olur?
 - a. İlk: 1, Orta: 4, Son: 7
 - b. İlk: 1, Orta: 3, Son: 7
 - c. İlk: 1, Orta: 2, Son: 3
 - d. İlk: 4, Orta: 5, Son: 6
 - e. 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 yapılırken 2. karşılaştırma adımında ilk, orta ve son elemanların konum bilgileri ne olur?
 - a. İlk: 1, Orta: 2, Son: 3
 - b. İlk: 1, Orta: 2, Son: 4
 - c. İlk: 1, Orta: 4, Son: 7
 - d. İlk: 4, Orta: 6, Son: 7
 - e. İlk: 5, Orta: 6, Son: 7
9. Ardışık arama algoritmasının en kötü durumdaki zaman karmaşıklığı değeri nedir?
 - a. $O(n)$
 - b. $O(n^2)$
 - c. $O(\log(n))$
 - d. $n.O(\log(n))$
 - e. $O(1)$
10. İkili arama algoritmasının en kötü durumdaki zaman karmaşıklığı değeri nedir?
 - a. $O(\log(n))$
 - b. $O(n^2)$
 - c. $O(n)$
 - d. $n.O(\log(n))$
 - e. $O(1)$

Kendimizi Sınavalım Yanıt Anahtarı

1. e Yanıtınız yanlış ise “Ardışık Arama” konusunu yeniden gözden geçiriniz.
2. e Yanıtınız yanlış ise “Ardışık Arama” konusunu yeniden gözden geçiriniz.
3. e Yanıtınız yanlış ise “Ardışık Arama” konusunu yeniden gözden geçiriniz.
4. b Yanıtınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
5. b Yanıtınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
6. c Yanıtınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
7. c Yanıtınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
8. e Yanıtınız yanlış ise “İkili Arama” konusunu yeniden gözden geçiriniz.
9. a Yanıtınız yanlış ise “Arama Algoritmalarının Karşılaştırılması” konusunu yeniden gözden geçiriniz.
10. a Yanıtınız 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 “büyüktür” 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("\nİkinci arama islemi:\n");
    ikili_arama(dizi, boyut, aranan2);
    printf("\nÜçüncü 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 vasıtasıyla 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);
}

/* İkili arama fonksiyonu */
void ikili_arama(int dizi[], int boyut, int aranan)
{
    int i, ilk, son, orta, karsilastirma_sayisi = 0;
    ilk = 0;
    son = boyut - 1;
    orta = (ilk + son) / 2;

    while (ilk <= son)
    {
        karsilastirma_sayisi++;
        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);

    printf("\nİkili 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);

    // İkili 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



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), yığın 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 karşılaştırılması ve gerekli durumlarda komşu elemanların yer değiştirmesine dayanır. Seçmeli sıralama, küçükten büyüğe doğru yapılacak bir sıralama işleminde her defasında dizinin en küçük elemanının tespit edilmesi ve uygun konuma yerleştirilmesine dayanır. Araya sokarak sıralama, diziden seçilen bir elemanın diğer elemanların kaydırılması vasıtasıyla 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. Yığın sıralaması, temel olarak dizinin elemanlarının bir yığın veri yapısı üzerinde temsil edilir hale getirilmesine ve sonrasında sıralanmasına dayanır. Ünitemizin ilerleyen bölümlerinde öncelikle temel sıralama algoritmaları örnekler yardımıyla detaylı olarak açıklanacak ve daha sonra C program kodu ile nasıl gerçekleştirildikleri gösterilecektir.

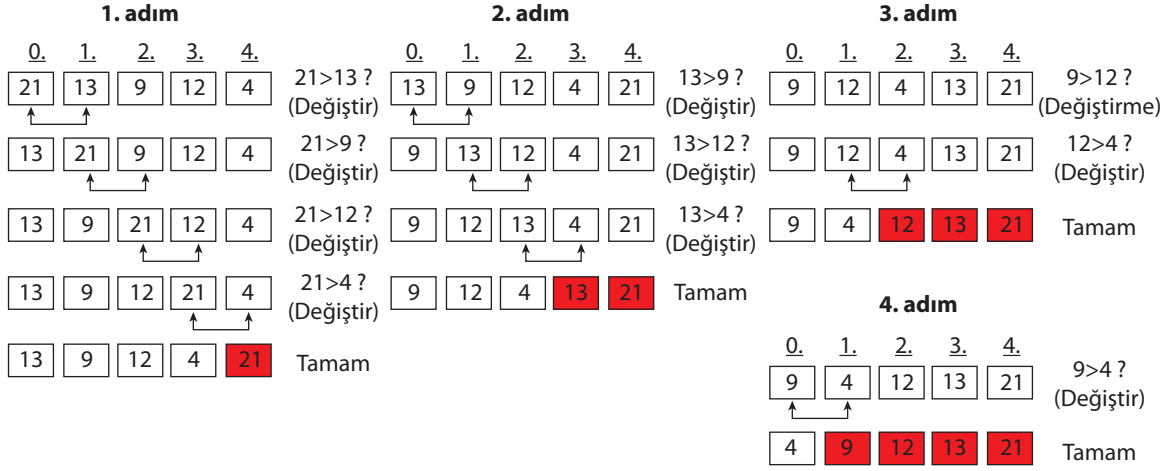
BALONCUK SIRALAMASI

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 karşılaştırmaya bir sonraki konumdaki dizi elemanı ile devam edilir. Bu işlem dizinin son elemanına ulaşıncaya 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 sona konumlandığı görülebilir. Her adımın bitiminde en sona konumlandırılan dizi elemanları, şekilde kırmızı ile işaretlenmiştir. Algoritmanın çalışması sırasında yapılan karşılaştırmalara da şeklin içerisinde yer verilmiştir. Algoritmanın 2. adımının sonunda, dizinin en büyük ikinci elemanı olan 13 sayısının sondan bir önceki konuma yerleştiği 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 sürdürülmüş ve algoritma 4 adımda tamamlanmıştır. Sıralamanın her bir sonraki adımında yapılan karşılaştırma işlemleri sayısının azaldığı 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 yapılmasına gerek duyulmamaktadır. 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_siralamasi.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdıran fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
}
```

```

    printf("\n");
}

/* Baloncuk siralamasi fonksiyonu */
void baloncuk_siralamasi(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("\nAdim %d.%d: ", i + 1, j + 1);
            dizi_yazdir(dizi, boyut);
        }
    }
    printf("-----\n");
    printf("\nSirali: ");
    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");

    baloncuk_siralamasi(sirasiz_dizi, boyut);
    getch();
}

```

Program kodları, *main* fonksiyonu ile çalışmaya başlamakta ve değerleri önceden belirlenmiş 5 elemanlı bir dizinin *baloncuk_siralamasi* 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_yazdir* isimli fonksiyon ise dizilerin elemanlarını ekrana yazdırmak için kullanılan yardımcı bir fonksiyondur. *baloncuk_siralamasi.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

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

Sirali: 4 9 12 13 21

SIRA SİZDE



baloncuk_siralamasi.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyı verdiğini kontrol ediniz.

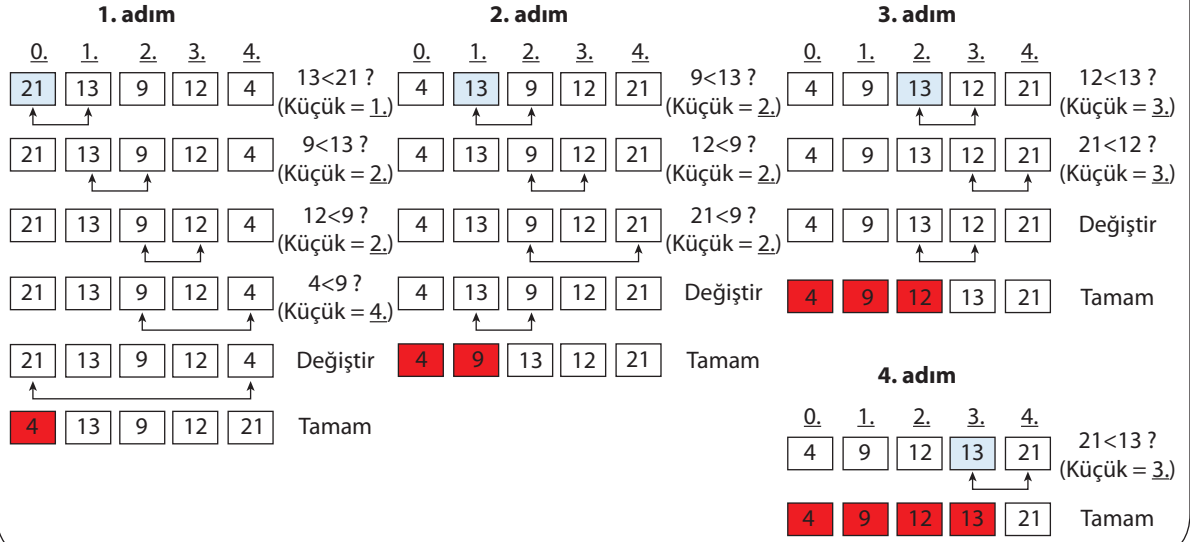
- Sıralama, küçükten büyüğe yerine büyüğe doğru yapılacaktır.
- *sirasiz_dizi* değişkeni içerisinde mevcut 5 elemanın yerine {9, 18, 12, 1} değerleri yer alacaktır.

SEÇMELİ SIRALAMA

Seçmeli sıralama, küçükten büyüğe doğru sıralama yapılacağı zaman adım adım dizilerin 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 varsayılır 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 Sıralama Algoritmasının Çalışmasına Bir Örnek



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. adımının 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 şeklin içerisinde yer verilmiştir. Her adımın bitiminde sıralı duruma gelen dizi elemanları kırmızı ile işaretlenmiştir. Algoritmanın 2. adımının 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. adımının 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 adımından 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 yazdıran 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("\nSirali: ");
    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.

```
Sirasiz: 21  13  9  12  4
-----
Yer degistirdi: [ 21, 4 ]

1. adim: 4  13  9  12  21
Yer degistirdi: [ 13, 9 ]

2. adim: 4  9  13  12  21
Yer degistirdi: [ 13, 12 ]

3. adim: 4  9  12  13  21

4. adim: 4  9  12  13  21
-----

Siralı: 4  9  12  13  21
```

`secmeli_siralama.c` program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyı 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

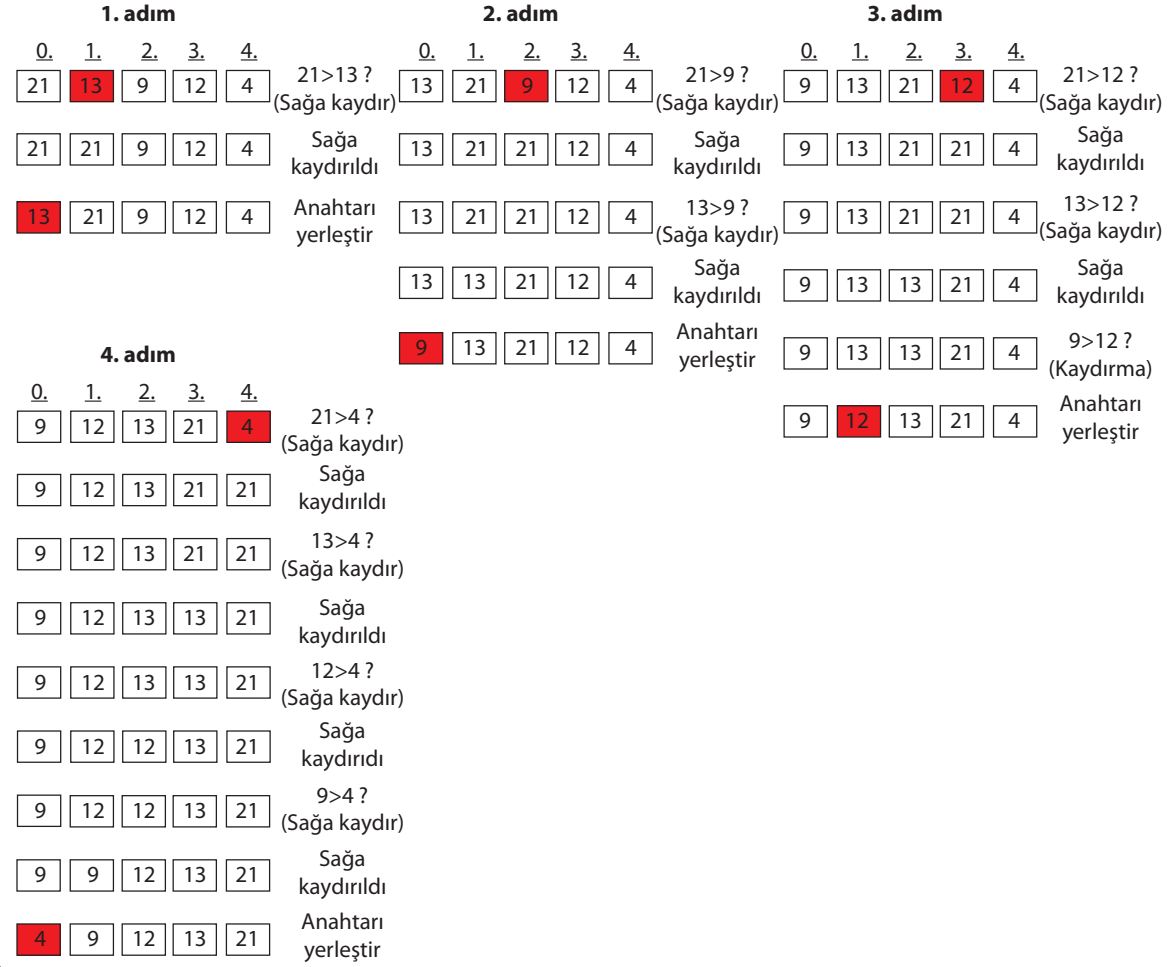
2

ARAYA SOKARAK SIRALAMA

Araya sokarak sıralama algoritması, dizinin elemanlarının kendilerinden önce gelen elemanlarla karşılaştırılması ve gerektiğinde birbirleriyle yer değiştirmeleri prensibine dayanır. Her bir adımda (iterasyonda), dizi elemanları üzerinde soldan sağa doğru hareket edilerek, kendisinden önce gelenlerle karşılaştırılacak bir anahtar eleman seçilir. Bu anahtar eleman, kendisinden önce gelen diğer tüm elemanlarla sırayla karşılaştırılır. 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 öncekiler kendisinden 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 varsayalım. Bu kağıdı kendisinden öncekilerle karşılaştırır ve ondan daha büyükleri sağa doğru kaydırırız. Daha sonra, çıkardığımız ve elimizde tuttuğumuz kağıdı uygun boşluğa yerleştiririz. Dolayısıyla, o oyun kağıdını yerleştirdiğimiz noktadan sonraki kağıtlar birer birer sağa kaymış olacaktır. Bu algoritmada, 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



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 elemanla 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 sayılardan 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 edildiği 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 yazdıran 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 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();
}

```

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.

```
Sirasiz: 21  13  9  12  4
-----

1. adim: 13  21  9  12  4

2. adim: 9   13  21  12  4

3. adim: 9   12  13  21  4

4. adim: 4   9   12  13  21
-----

Siralı:  4   9   12  13  21
```

SIRA SİZDE



3

araya_sokarak_siralama.c program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyı verdiğini kontrol ediniz.

- **Sıralama, küçükten büyüğe yerine büyüğe doğru yapılacaktır.**
- *sirasiz_dizi* değişkeni içerisinde mevcut 5 elemanın yerine {9, 18, 12, 1} değerleri yer alacaktır.

HIZLI SIRALAMA

Hızlı sıralama algoritması, şimdiye 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 içerisinden 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 gruba ayrılır. Daha sonra, aynı işlemler dizinin iki grubu için bağımsız olarak tekrarlanır. Algoritmanın çalışması tamamlandığında, kendi içerisinde sıralı iki dizinin birleştirilmesiyle tek bir sıralı dizi elde edilir. Şekil 7.4'te hızlı sıralama algoritmasının çalışmasına bir örnek verilmiştir.

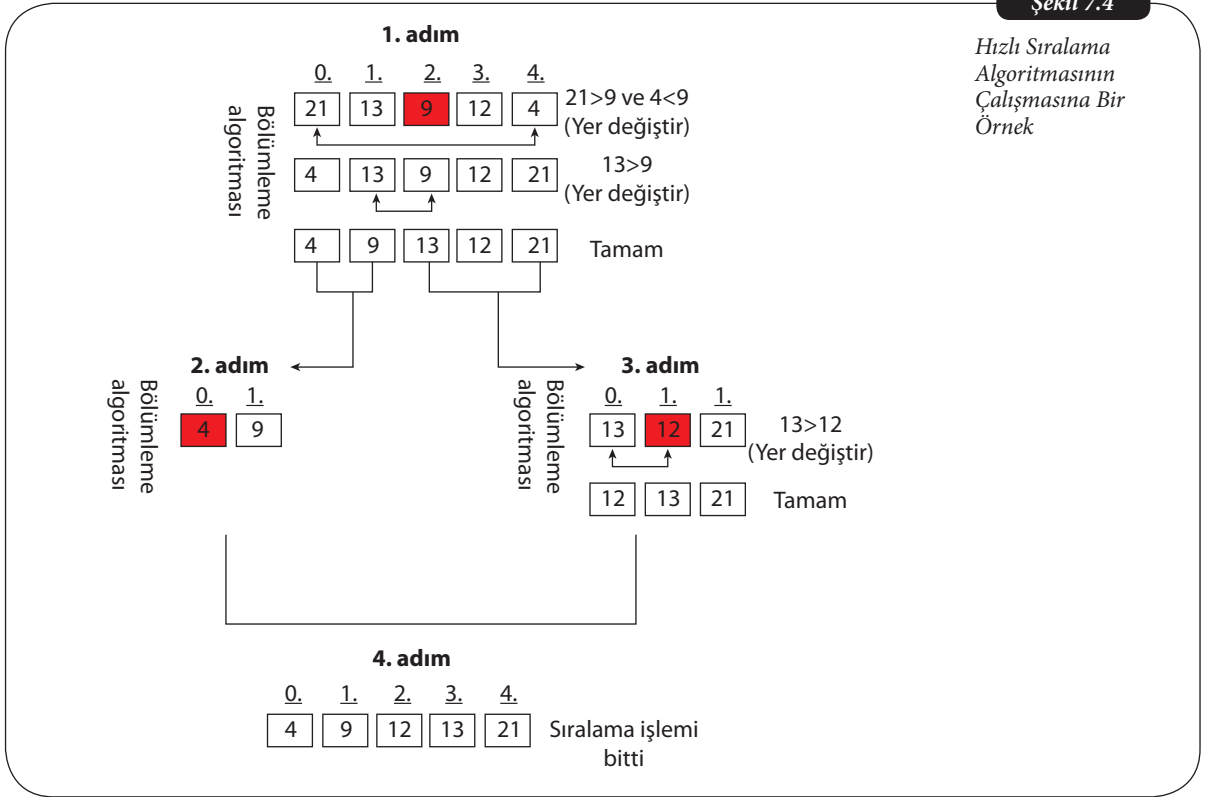
Özyinelemeli (Recursive)

Fonksiyon: Özyinelemeli fonksiyonlar, kendi içlerinde tekrar kendilerini çağıran 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 bölümlenme için seçilen sınır değeridir. Hızlı sıralama algoritması, pivot elemandan küçük ve büyük değerleri iki farklı gruba ayırmaya çalışır.

Şekil 7.4

Hızlı Sıralama Algoritmasının Çalışmasına Bir Örnek



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, bölümlenme 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 birbiriyle 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 itibarıyla, 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 bölümlenme 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 edildiği Örnek 7.4'te verilmiştir.

Hızlı sıralama algoritmasına yönelik örnek kod

ÖRNEK 7.4

```
/* hizli_siralama.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdıran 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 bölümlenme ve 2 defa `hizli_siralama` fonksiyonunu çağırdığı görülmektedir. Dolayısıyla, `hizli_siralama` fonksiyonu özyinelemeli bir fonksiyondur. Şekil 7.4'te de gösterildiği gibi bölümlenme 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 bölümlenme fonksiyonunu çağırmasıyla dizinin sıralanması söz konusu olur. `dizi_yazdir` isimli fonksiyon ise dizilerin elemanlarını yazdırmak için kullanılan yardımcı bir fonksiyondur. `hizli_siralama.c` uygulaması çalıştırıldığında aşağıdaki ekran görüntüsü elde edilir.

Sirasız: 21 13 9 12 4

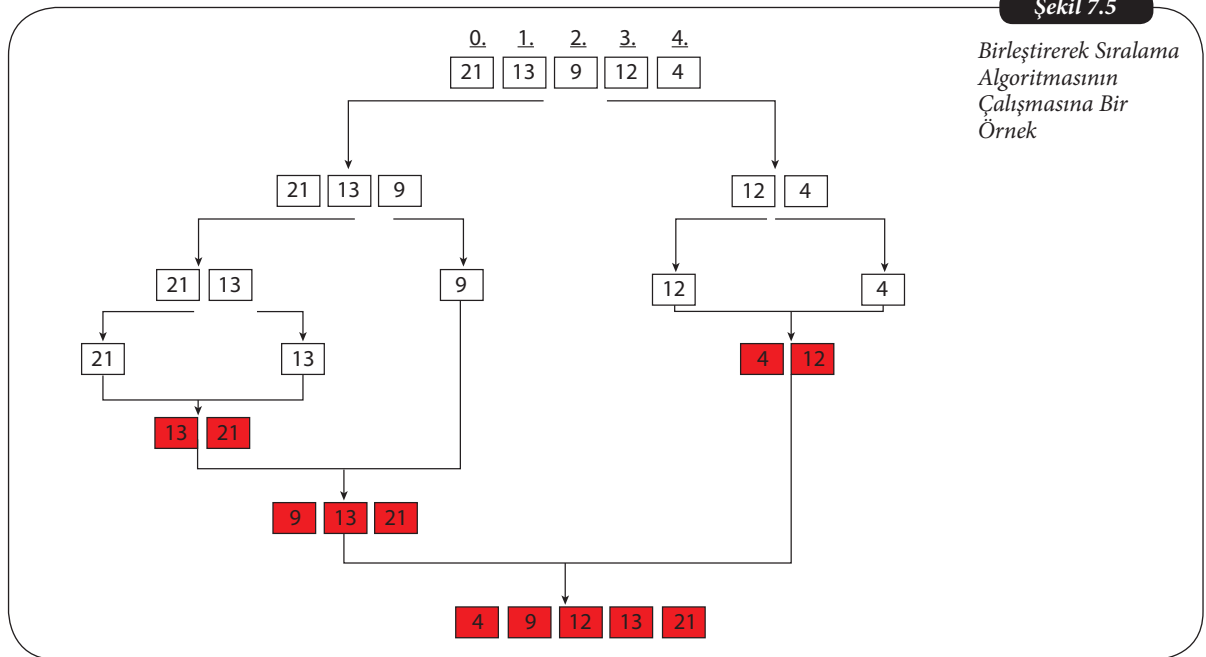
Sıralı: 4 9 12 13 21

`hizli_siralama.c` program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyı 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.

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 oluşturulurken herhangi bir sayıdan küçük veya büyük şeklinde bir ayrıma 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.



Birleştirmeli 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 diziyeye 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 diziyeye 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 yazdıran 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 sirali olarak birlestiriliyor*/
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 ayrilarak 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("\nSiralı: ");
    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
-----

Siralı:  4  9  12  13  21

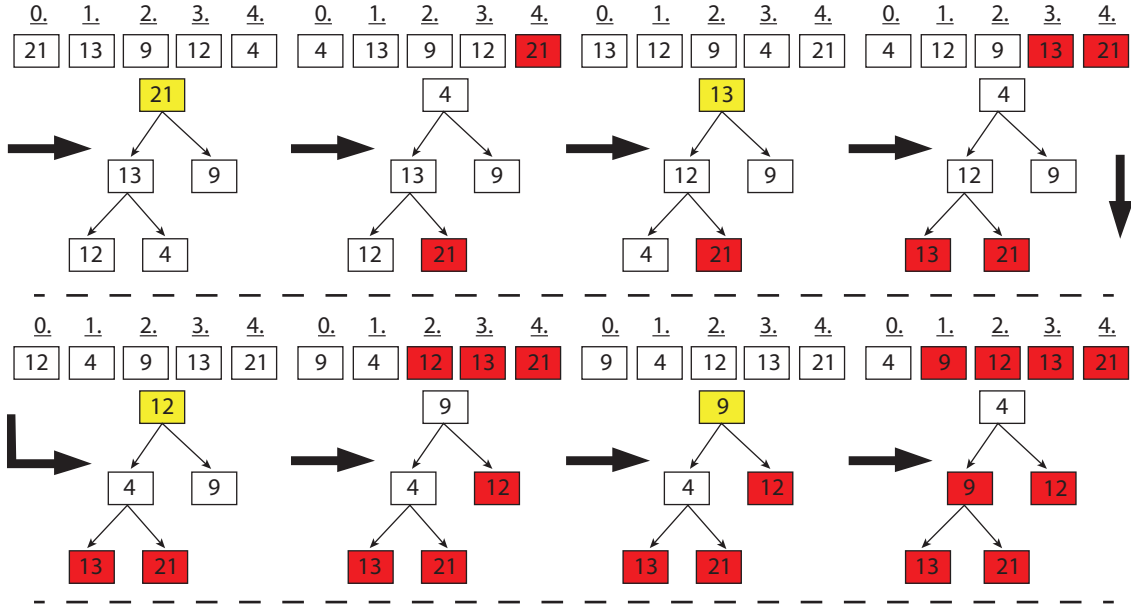
```

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 verilmiştir. 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 yığınin en büyük elemanı bulunmakta ve sarı ile gösterilen bu eleman, ağacın kök konumuna yerleştirilmektedir. Dolayısıyla, maksimum yığın yapısının kullanıldığı söylenebilir. Her bir elemanın 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 adımından 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 edildiği Örnek 7.6'da verilmiştir.

ÖRNEK 7.6

Yığın sıralama algoritmasına yönelik örnek kod

```

/* yigin_siralamasi.c */

#include <stdio.h>

/* Verilen diziyi ekrana yazdıran 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_siralamasi(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_siralamasi* isimli C fonksiyonuna parametre olarak gönderilmektedir. isimli C fonksiyonu, kendi içerisinde 2 adet *maksimuma_yiginla* fonksiyonunu çağırılmaktadır. İlk *maksimuma_yiginla* çağırıldığında, dizi yığın halinde temsil edilmektedir. Daha sonra *for* döngüsü içerisinde çağrılan *maksimuma_yiginla* fonksiyonu vasıtasıyla, 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
-----
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
-----
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 varsayalım. 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 yığın sıralamasının diğer sıralama algoritmalarına göre daha düşük zaman karmaşıklığına sahip olduğu anlaşılmaktadır. İstikrarlılık açısından değerlendirme yapıldığında ise yığın 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	$O(n \cdot \log(n))$	Evet
Yığın sıralaması	$O(n \cdot \log(n))$	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 kavramlar olması mümkündür. Ünitimizde, 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 algoritmalarlardır.



Temel sıralama algoritmalarını uygulamak

Bu ünite, 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), yığın 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 yığın sıralaması algoritmaları, açıklanan diğer sıralama algoritmalarından farklı olarak özyinelemeli fonksiyonlar yardımıyla gerçekleştirilmektedir.



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 yığın sıralaması algoritmalarının, sahip oldukları $O(n \cdot \log(n))$ karmaşıklık seviyesi ile diğerlerinden daha hızlı çalışmalarının mümkün olduğu söylenilebilir. Çünkü baloncuk sıralaması, seçmeli sıralama, araya sokarak sıralama ve hızlı sıralama algoritmaları $O(n^2)$ karmaşıklık seviyesine sahiptir. Bir algoritmanın istikrarlı olması, dizi içerisinde eşit elemanlar bulunması durumunda, sıralama sonrasında bu elemanların birbirlerine göre bağıl olarak yerlerinin değişmemesi anlamına gelir. Bu açıdan, seçmeli sıralama ve yığın 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 Sıyalım

1. Elemanları [7, 6, 9, 1] olan bir dizi, baloncuk sıralaması algoritması ile küçükten büyüğe doğru sıralanmak istenildiğinde, algoritmanın adımları sonrasında elde edilecek diziler aşağıdaki seçeneklerin 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 aşağıdaki seçeneklerin 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 aşağıdaki seçeneklerin 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 aşağıdaki seçeneklerin 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 bölümlenme algoritması 1 defa çalışıp tamamlandıktan sonra dizinin son durumu aşağıdaki seçeneklerden 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 bölümlenme algoritması 1 defa çalışıp tamamlandıktan sonra dizinin son durumu aşağıdaki seçeneklerden 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 aşağıdaki seçeneklerin 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 aşağıdaki seçeneklerin 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 durumdaki 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 Yanıtınız yanlış ise “Baloncuk Sıralaması” konusunu yeniden gözden geçiriniz.
2. c Yanıtınız yanlış ise “Baloncuk Sıralaması” konusunu yeniden gözden geçiriniz.
3. b Yanıtınız yanlış ise “Seçmeli Sıralama” konusunu yeniden gözden geçiriniz.
4. a Yanıtınız yanlış ise “Seçmeli Sıralama” konusunu yeniden gözden geçiriniz.
5. a Yanıtınız yanlış ise “Hızlı Sıralama” konusunu yeniden gözden geçiriniz.
6. e Yanıtınız yanlış ise “Hızlı Sıralama” konusunu yeniden gözden geçiriniz.
7. d Yanıtınız yanlış ise “Araya Sokarak Sıralama” konusunu yeniden gözden geçiriniz.
8. a Yanıtınız yanlış ise “Araya Sokarak Sıralama” konusunu yeniden gözden geçiriniz.
9. e Yanıtınız yanlış ise “Sıralama Algoritmalarının Özellikleri” konusunu yeniden gözden geçiriniz.
10. c Yanıtınız yanlış ise “Sıralama Algoritmalarının Özellikleri” konusunu yeniden gözden geçiriniz.

Sıra Sizde Yanıt Anahtarı

Sıra Sizde 1

İstenilen değişiklikler yapıldıktan sonra *baloncuk_siralamasi.c* programının kodları aşağıdaki gibi olmalıdır. *baloncuk_siralamasi* fonksiyonunun içerisindeki karşılaştırma operatörünün, büyüktür yerine küçüktür olarak güncellenmesi gerekmektedir. Ayrıca, dizinin içerisindeki sayıların değiştirilmesi istenildiği için *sirasiz_dizi* isimli değişkenin içeriği güncellenmelidir.

```
/* baloncuk_siralamasi.c */
#include <stdio.h>

/* Verilen diziyi ekrana yazdıran fonksiyon */
void dizi_yazdir(int dizi[], int boyut)
{
    int i;
    for (i = 0; i < boyut; i++)
    {
        printf("%d\t", dizi[i]);
    }
    printf("\n");
}

/* Baloncuk siralamasi fonksiyonu */
void baloncuk_siralamasi(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("\nAdim %d.%d: ", i + 1, j + 1);
            dizi_yazdir(dizi, boyut);
        }
        printf("-----\n");
        printf("\nSiralı: ");
        dizi_yazdir(dizi, boyut);
    }
}

void main()
{
    int aranan;
    int sirasiz_dizi[] = { 9, 18, 12, 1 };

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

    printf("\nSirasiz: ");
    dizi_yazdir(sirasiz_dizi, boyut);
    printf("-----\n");

    baloncuk_siralamasi(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, küçüktür yerine büyüktür 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 yazdıran 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, enbuyukyer, gecici;
    for (i = 0; i < boyut - 1; i++)
    {
        // ilk elemani en buyuk olarak ata
        enbuyukyer = i;
        // dizinin o anki en buyuk elemanini 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 buyuk elemani i. eleman ile degistir
            int gecici = dizi[enbuyukyer];
            dizi[enbuyukyer] = dizi[i];
            dizi[i] = gecici;
        }
        printf("\n%d. adim: ", i + 1);
        dizi_yazdir(dizi, boyut);
    }
    printf("-----\n");
    printf("\nSiralı: ");
    dizi_yazdir(dizi, boyut);
}
void main()
{
    int aranan;
    int sirasiz_dizi[] = { 9, 18, 12, 1 };
    // 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();
}

```


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 güncellenmelidir.

```

/* araya_sokarak_siralama.c */

#include <stdio.h>

/* Verilen diziyi ekrana yazdıran 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. adim: ", 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 yazdıran 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;
}

/* Hizli siralama 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 sayisi 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();
}

```

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

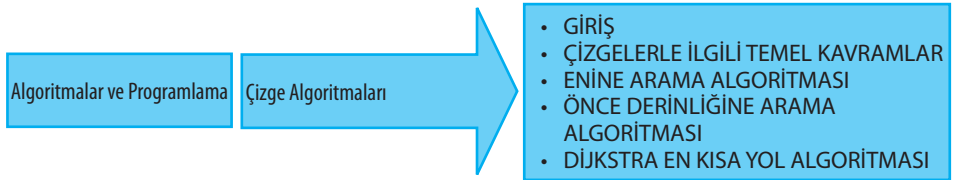
Bu üniteyi tamamladıktan sonra;

- 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



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 hayattaki birçok alanda uygulamaları mevcuttur. Örneğin, bir sosyal ağ uygulamasını göz önüne aldığımızda, 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 içerdiği 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 yanı sıra, çizgelerin içerdiği düğüm noktaları arasındaki en kısa mesafelerin hesaplanması da söz konusu olabilir. Örneğin, bir şehirden başka bir şehire araçla en hızlı şekilde ulaşım istenildiğini varsayalım. 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. Ünitimizde, bu tipte en yaygın algoritmalarından 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. Ünitimizin 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{G} ifadesi çizgeyi, D ve K ifadeleri ise sırasıyla düğümleri ve kenar bağlantılarını temsil etmektedir.

$$\mathcal{G} = (D, K)$$

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):

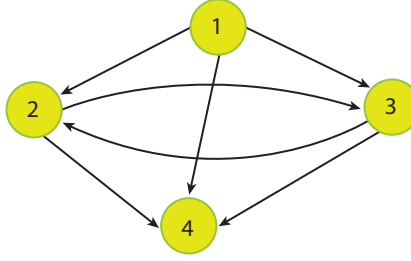
Yönlü çizge, kenar bağlantılarının yönleri temsil eden oklarla gösterildiği çizgedir. Çizge içerisindeki birbirine bağlı iki düğüm noktası arasında, sadece ilgili okun işaret ettiği yönde ilerlenebilmesi mümkündür.

Yönsüz çizge (undirected graph):

Yönsüz çizge, kenar bağlantılarının yönleri temsil eden oklar ile gösterilmediği 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.

Şekil 8.1

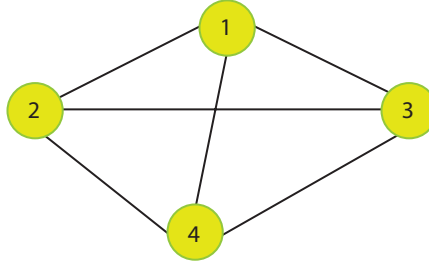
Yönlü Çizge Örneği



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.

Şekil 8.2

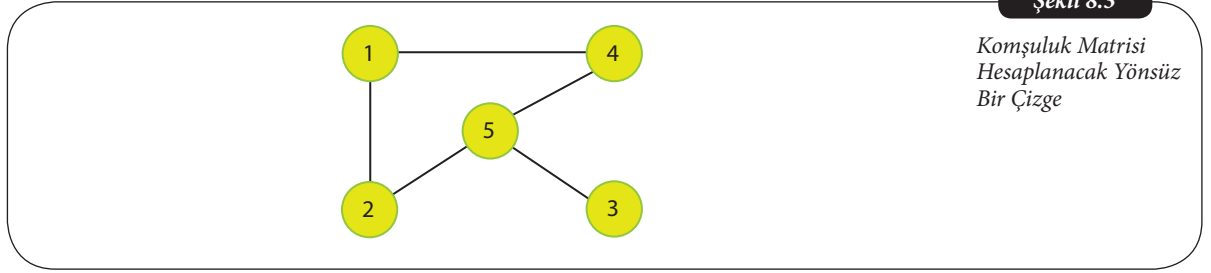
Yönsüz Çizge Örneği



Şekil 8.2'de, kenar bağlantıları oklar kullanılmadan temsil edilmektedir. Dolayısıyla, bu çizgenin yönsüz bir çizge olduğu anlaşılabilir. Şekildeki içerik, küme gösterimleri kullanılarak $D=\{1, 2, 3, 4\}$ ve $K=\{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$ 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. Çizgeler hakkındaki bir diğer önemli husus ise düğümlerin komşuluk ilişkilerinin temsil edilme şeklidir. Bu amaçla kullanılacak 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 döğümler için 1 değeri kullanılırken, komşuluk olmayan döğümler için 0 değeri kullanılmaktadır. İlk satırda iki adet değerin 1 olmasının sebebi, 1 döğümünün 2 ve 4 döğümleri ile komşu olmasıdır. Başka bir komşuluk olmadığı için ilk satırdaki diğer değerler 0 ile gösterilmektedir. Son satırda ise üç adet değerin 1 olmasının sebebi, 5 döğümünün 2, 3 ve 4 döğümleri ile komşu olmasıdır. Diğer döğümlerle 5 döğümü 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.

Döğüm	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 matrister, döğümler arasında kenar bağlantılarının bulunup bulunmadığını göstermek için 0 ve 1 değerleri kullanılmıştır. Bu örnekte, çizgenin kenar bağlantıları sayısal olarak ağırlıklandırılmamıştır 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.

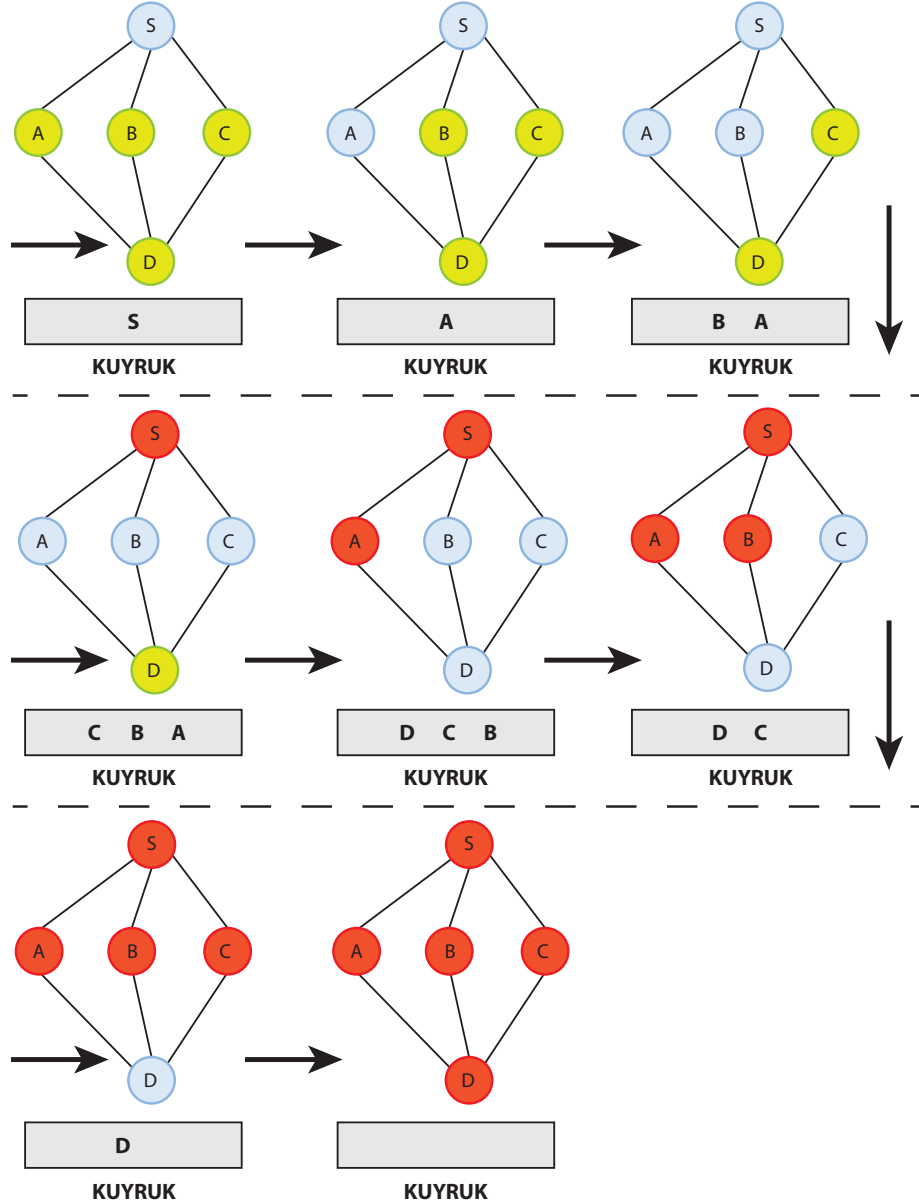
ENİNE ARAMA ALGORİTMASI

Enine arama, çizgenin bir döğümünden başlanarak, söz konusu döğümün komşu döğümlerinin ve onların da komşularının sırayla ziyaret edildiği arama algoritmasıdır. Bu algoritmanın çalışması sırasında, öncelikle başlangıç döğümünün tüm komşuları ziyaret edilir. Daha sonra, başlangıç döğümünün komşuları ile komşu olan döğümlerden devam edilir. Algoritmanın uygulanması esnasında kuyruk (queue) veri yapısından faydalanılır. Şekil 8.4'te enine arama algoritmasının çalışmasına bir örnek verilmiştir. Bu örnekte, bir döğümün birden fazla komşusu olduğunda bu komşular alfabetik sırada küçükten büyüğe doğru ziyaret edilmektedir.

Ağırlıklandırılmış çizge (weighted graph): Döğümler arasındaki kenar bağlantıları üzerinde sıfırdan farklı sayısal değerlerin yer aldığı çizge türüdür.

Şekil 8.4

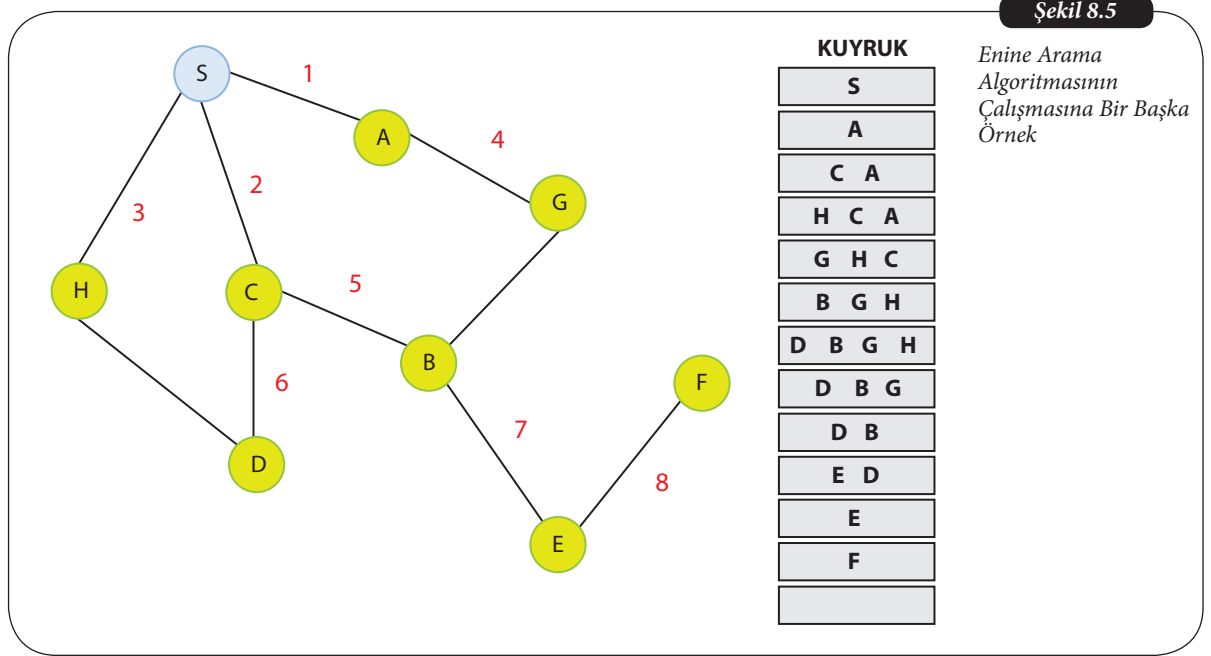
Enine Arama
Algoritmasının
Çalışmasına Bir
Örnek



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 yapılırken kuyruk veri yapısından faydalanılmaktadır. 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. Algoritmaya 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 hepsinin 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;

// çizge degiskenleri

// dugumler dizisi
struct Dugum* dugumlistesi[MAKSIMUM];
// komsuluk matrisi
int komsulukmatrisi[MAKSIMUM][MAKSIMUM];
// dugum noktasi sayisi
int dugumsayisi = 0;

//kuyruk fonksiyonlari
void kuyruge_ekle(int veri)
{
    kuyruk[++arkasi] = veri;
    kuyrukelemanisayisi++;
}

int kuyruktan_cikar()
{
    kuyrukelemanisayisi--;
    return kuyruk[onu++];
}

bool kuyruk_bos()
{
    return kuyrukelemanisayisi == 0;
}

// çizge 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 bağlantısı ekle
void yonsuz_kenarbaglantisi_ekle(int baslangic, int bitis)
{
    komsulukmatrisi[ baslangic][ bitis] = 1;
    komsulukmatrisi[ bitis][ baslangic] = 1;
}

// yeni tek yonlu kenar bağlantısı ekle
void yonlu_kenarbaglantisi_ekle(int baslangic, int bitis)
{
    komsulukmatrisi[ baslangic][ bitis] = 1;
}

// dugum görüntüle
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 ]->ziyaretedildimi == false)
            return i;
    }
    return -1;
}

void enine_arama_algoritmasi()
{
    int i;

    // baslangic dugumunun durumunu ziyaret edildi olarak isaretle
dugumlistesi[ 0 ]->ziyaretedildimi = true;

    //dugum görüntüle
dugum_goruntule(0);

    // dugum indeksini kuyruğa ekle
kuyruğa_ekle(0);
    // ziyaret edilmemiş dugum
int ziyaretedilmemisdugum;

    while (!kuyruk_bos())
    {
        // islem yapılacak ziyaret edilmemis dugumu getir
int islemyapilandugum = kuyruktan_cikar();

        // islem yapılan dugumun tum komsu dugumlerini ziyaret et
while ((ziyaretedilmemisdugum =
ziyaretedilmeyen_komsuyu_getir(islemyapilandugum)) != -1)
        {
            dugumlistesi[ ziyaretedilmemisdugum] -
>ziyaretedildimi = true;
            dugum_goruntule(ziyaretedilmemisdugum);
            kuyruğa_ekle(ziyaretedilmemisdugum);
        }
    }
}

```

```

        // kuyruk 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;
        }
        // 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 Sonuclari: " );
        enine_arama_algoritmasi();
        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 *kuyruğa_ekle* ve *kuyruktan_cikar* gibi fonksiyonlar bulunmaktadır. Bir diğer fonksiyon olan *kuyruk_bos* ise kuyruğun içerisinde 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_algoritmasi* fonksiyonu sayesinde gerçekleştirilmektedir. Bu fonksiyon, çalışmaya en baştaki düğümün ziyaret edilmesi ve kuyruk veri yapısı içerisine *kuyruğa_ekle* fonksiyonu ile eklenmesiyle başlamaktadır. Daha sonra, *while* döngüsü içerisindeki kod bloğu çalışmakta ve kuyruқта eleman bulunduğu sürece, *kuyruktan_cikar* fonksiyonu vasıtasıyla 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 çıktıyı verdiğini kontrol ediniz.

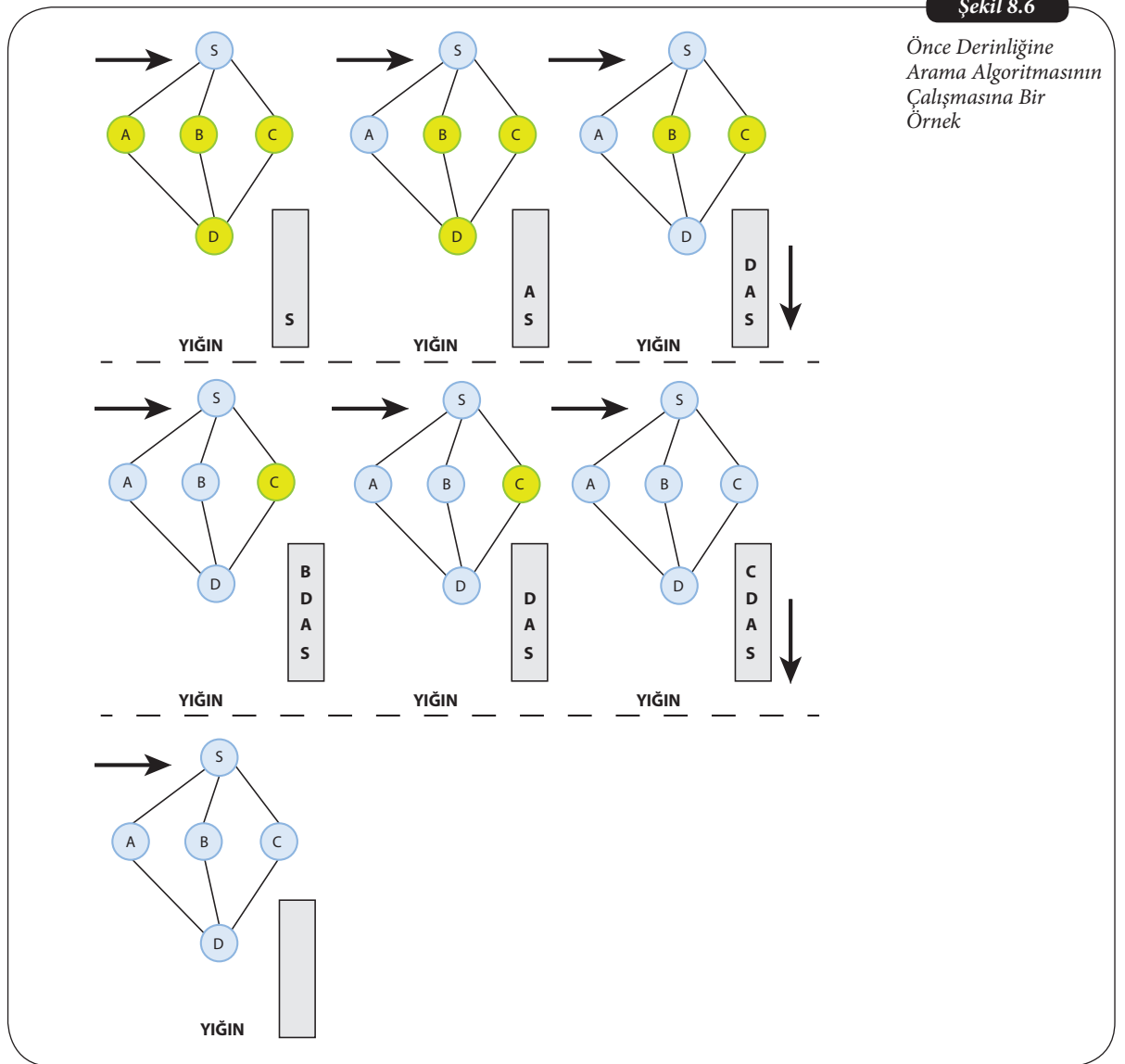
- **MAKSIMUM** isimli değişkenin boyutunu Şekil 8.5'teki toplam düğüm sayısını göz önüne alarak güncelleyiniz.
- **main** fonksiyonunun içeriğini Şekil 8.5'teki örnek çizge için çalışacak şekilde düzenleyiniz ve enine arama sırasında çizgenin düğümlerinin ziyaret sıralamasının ekranda nasıl sonuç verdiğini gözleyiniz.



SIRA SİZDE

ÖNCE DERİNLİĞİNE ARAMA ALGORİTMASI

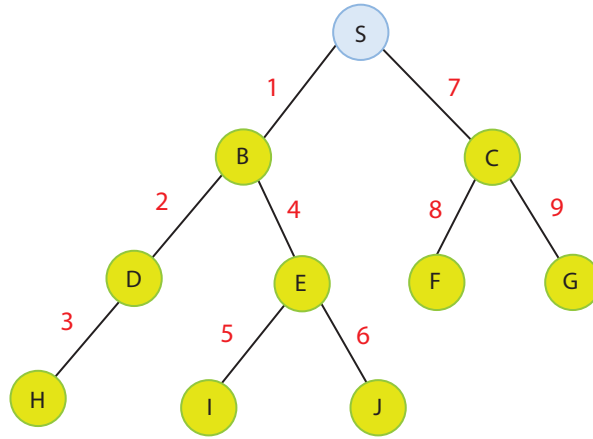
Önce derinliğine arama, çizgenin bir düğümünden başlanarak bu düğümün komşusu üzerinden gidilebilecek en uzak düğüme kadar olan noktaların ziyaret edildiği ve daha sonra geri dönülerek aynı işlemlerin ziyaret edilmemiş düğümler için sürdürüldüğü bir arama algoritmasıdır. Algoritmanın uygulanması esnasında yığın (stack) veri yapısından faydalanılır. Şekil 8.6'da enine arama algoritmasının çalışmasına bir örnek verilmiştir.



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 yapılırken yığın veri yapısından faydalanılmakta olup, yığın veri yapısının anlık içeriğine şekilde yer verilmiştir. Algoritma ilk çalıştığında aramaya çizgenin S isimli düğümünden başlanmıştır ve S düğümü yığının içerisine eklenerek mavi olarak gösterilmiştir. Daha sonra, komşular arasında alfabetik olarak en önde gelen A düğümü ile devam edilmiştir. Bu düğüm de yine yığına eklendikten sonra A'nın komşusu olan D düğümüyle devam edilmiştir. Bu aşamada, D düğümü de yığına eklenmiş ve şekilde mavi hale getirilmiştir. D düğümünün komşuları B ve C düğümleridir. Alfabetik sıra göz önüne alındığında B düğümü ile ziyarete devam edilmiştir ve B düğümü de yığına eklenmiştir. Bu aşamadan sonra, B düğümünün ziyaret edilmemiş komşusu bulunmadığı için bu düğüm yığından çıkarılmış ve bir önceki düğüme geri dönmüştür. Bir önceki düğüm olan D düğümünün ziyaret edilmemiş komşusu durumundaki C düğümü ziyaret edilmiş ve yığına eklenmiştir. Bu işlemlerin sonunda tüm düğümler ziyaret edilmiş durumdadır. Sonrasında, yığındaki elemanlar tek tek çıkarılmakta ve çıkarılanın ilgili düğümün ziyaret edilmemiş komşusu olup olmadığına bakılmaktadır. Düğümlerin tamamı ziyaret edilmiş durumda olduğundan, yığındaki tüm elemanlar sırayla çıkarılmıştır. Son olarak, yığında eleman 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

Önce Derinliğine
Arama Algoritmasının
Çalışmasına Bir Başka
Örnek



Şeklin üzerindeki kırmızı sayılar, algoritmanın çalışması esnasında 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 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 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 kuyruğa ekle
yigina_ekle(0);
    // ziyaret edilmemiş dugum
int ziyaretedilmemisdugum;
while (!yigin_bos())
{
    // ziyaret edilmemiş kuyruğu getir
ziyaretedilmemisdugum = ziyaretedilmeyen_komsuyu_
getir(yigin_ensusstekini_goster());

    if (ziyaretedilmemisdugum == -1)
    {
        yigindan_cikar();
    }
    else
    {
        dugumlistesi[ ziyaretedilmemisdugum] -
>ziyaretedildimi = true;
dugum_goruntule(ziyaretedilmemisdugum);
yigina_ekle(ziyaretedilmemisdugum);
    }
}
// yigin bos, arama tamamlandı, degiskenin degerini ilk haline
getir
for (i = 0; i<dugumsayisi; i++)
{
    dugumlistesi[i]->ziyaretedildimi = false;
}
}

```



```

void main()
{
    int i, j;
    // komşuluk matrisini başlangıçta sıfırla
    for (i = 0; i<MAKSIMUM; i++)
    {
        for (j = 0; j<MAKSIMUM; j++)
            komşulukmatrisi[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("\nÖnce Derinliğine Arama Sonuçları: ");
    once_derinligine_arama_algoritmasi();
    getch();
}

```

Önce derinliğine arama algoritmasının çalışma mantığının anlatıldığı Örnek 8.2'de, çizge ve yığın veri yapısı için kullanılan değişken ve fonksiyonlar bulunmaktadır. Öncelikle, yığın veri yapısı `yigin` isimli bir değişken kullanılarak gerçekleştirilmektedir. Şekil 8.6'da görülen yığına ekleme, çıkarma işlemleri için `yigina_ekle` ve `yigindan_cikar` gibi fonksiyonlar bulunmaktadır. Bir diğer fonksiyon olan `yigin_bos` ise yığının 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_algoritmasi` fonksiyonu sayesinde gerçekleştirilmektedir. Bu fonksiyon, çalışmaya en baştaki düğümün ziyaret edilmesi ve yığın veri yapısı içerisine `yigina_ekle` fonksiyonu ile eklenmesiyle başlamaktadır. Daha sonra, `while` döngüsü içerisindeki kod bloğu çalışmakta ve kuyrukta eleman bulunduğu sürece `yigindan_cikar` fonksiyonu vasıtasıyla üzerinde işlem yapılacak düğüm, yığından çıkarılarak belirlenmektedir. Öncelikle üzerinde işlem yapılan düğümde 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 yığından eleman çıkararak üst düğümlere doğru geri dönülmekte ve bu kez onların komşuları arasında ziyaret edilmemiş düğüm olup olmadığı incelenmektedir. Yığın veri yapısı içerisindeki tüm düğümler çıkarıldığında `yigin_bos()` fonksiyonunun yardımıyla `while` döngüsü sonlanmakta ve önce derinliğine arama algoritmasının çalışması tamamlanmaktadır.



`once_derinligine_arama.c` program kodlarını aşağıdaki bilgiler doğrultusunda tekrar düzenleyip çalıştırarak uygun çıktıyı verdiğini kontrol ediniz.

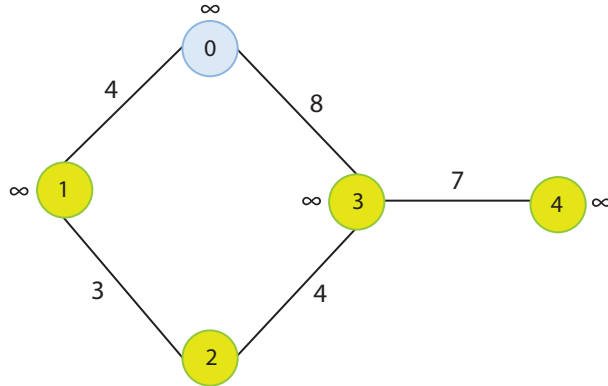
- **MAKSIMUM** isimli değişkenin boyutunu Şekil 8.7'deki toplam düğüm sayısını göz önüne alarak güncelleyiniz.
- **main** fonksiyonunun içeriğini Şekil 8.7'deki örnek çizge için çalışacak şekilde düzenleyiniz ve önce derinliğine arama sırasında çizgenin düğümlerinin ziyaret sıralamasının ekranda nasıl sonuç verdiğini 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 varsayalım. Düğümler arasındaki ağırlıklandırılmış kenar bağlantıları ise her iki şehir çifti arasındaki uzaklığı 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 varsayılır. Algoritma, düğümler arasındaki en kısa yolları aradığı için düğümler arasında daha kısa yollar bulunduğunda, sonsuz değeri ilgili yolun uzunluğu ile değiştirilir. Şekil 8.8'de Dijkstra algoritması ile en kısa yolun bulunacağı 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



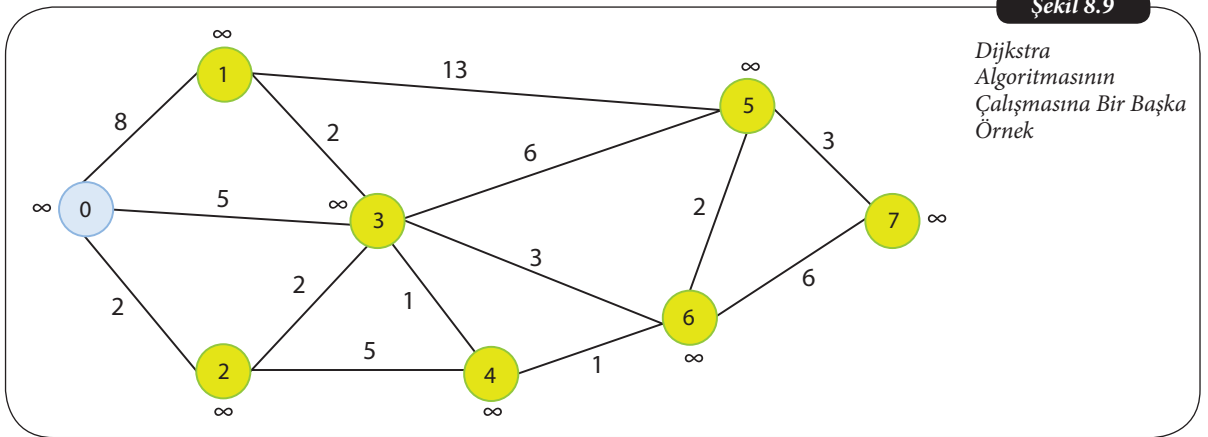
Dijkstra algoritması, her bir adımında ziyaret edilenler listesine yeni bir düğüm eklemekte ve düğümler arası mesafeleri güncellemektedir. Tablo 8.2'de algoritmanın adım adım ilerleyişi ve 0 düğümü ile diğerleri arasındaki mesafeler belirtilmektedir. Mesafe değeri kesinleşen ve üzerinde artık işlem yapılmayacak düğümleri gösteren hücreler, tablo içerisinde mavi olarak görüntülenmektedir. Ayrıca, parantez içerisinde belirtilen değerler, o mesafenin elde edilmesi için ziyaret edilmesi gerekli olan düğümü göstermektedir. Tabloda ilk olarak 0 düğümü ziyaret edilmiştir. 0 düğümü ile diğer düğümler 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 satırdaki mavi olmayan değerler içerisinde en küçük olan tespit

edilir. 4, 8 ve sonsuz değerleri içerisinde en küçüğü 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 küçüğü 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 mesafede bir değişiklik yapılmayacaktır. Böyle bir durumda, mesafe değeri olarak daha küçük bir sayı elde edilebilseydi tablonun güncellenmesi gerekirdi. Daha sonra, mevcut satırda mavi olmayan 8 ve sonsuz arasından en küçük olan değer tespit edilmesi gereklidir. 8 değeri küçük olduğu için bir sonraki adımda 3 düğümü ziyaret edilecektir. 3 düğümü ile 4 düğümü arasında komşuluk bulunduğu için tabloya 15 sayısı yazılmıştır. Bunun sebebi, 3 düğümüne ulaşmak için en kısa mesafenin 8 olduğunun 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)	∞ (0)	8(0)	∞ (0)
1	0(0)	4(0)	7(1)	8(0)	∞ (0)
2	0(0)	4(0)	7(1)	8(0)	∞ (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 yolun bulunacağı bir başka çizge örneği verilmiştir. Bu örnekte, 0 düğümü ile diğer düğümler arasındaki en kısa yol aranacağı için 0 düğümü mavi ile gösterilmektedir.



Şekil 8.9

Dijkstra
Algoritmasının
Çalışmasına Bir Başka
Örnek

Dijkstra algoritması, her bir adımında 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 gerekli 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 Srasındaki
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 ulaşabiliriz. 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)'tür. Bu bilgiden, bir adım önce 4 düğümüne uğranması gerektiği anlaşılmaktadır. Aynı şekilde, başlığı 4 olan sütun için son satır değeri 5(3)'tür. Bu bilgiden, bir adım önce 3 düğümüne uğranması gerektiği anlaşılmaktadır. Aynı şekilde, başlığı 3 olan sütun için son satır değeri 4(2)'tür. Bu bilgiden, bir adım önce 2 düğümüne uğranması gerektiği anlaşılmaktadır. Bunları bir araya getirdiğimizde ise 0

düğümünden 5 düğüme 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 ifadesi 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;
};

// çizge degiskenleri

// dugumler dizisi
struct Dugum* dugumlistesi[ MAKS ] ;
// komsuluk matrisi
int komsulukmatrisi[ MAKS ][ MAKS ] ;
// dugum noktasi sayisi
int dugumsayisi = 0;

// çizge 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 kısa yol algoritma fonksiyonu
    void dijkstra(int baslangicdugumu)
    {
        int mtablosu[ MAKS][ MAKS], mesafe[ MAKS], oncesi[ MAKS],
enkisayol[ MAKS];
        int ziyaretli[ MAKS], sayi, enkmesafe, sonraki, i, j, k;

        //oncesi[] her dugumun oncesindeki dugumu saklar
        //en kısa 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];
            oncesi[ i] = baslangicdugumu;
            ziyaretli[ i] = 0;
        }

        mesafe[ baslangicdugumu] = 0;
        ziyaretli[ baslangicdugumu] = 1;
        sayi = 1;

        while (sayi<MAKS - 1)
        {
            enkmesafe = SONSUZ;

            //sonraki en kısa 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];
                        oncesi[ i] = sonraki;
                    }

            sayi++;
        }

        //en kısa mesafenin elde edildiği 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 kısa 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;
    // komşuluk matrisini baslangicta sıfırla
    for (i = 0; i<MAKS; i++)
    {
        for (j = 0; j<MAKS; j++)
            komşulukmatrisi[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 bağlantılarını 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 algoritmasını çalıştır
    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 deęişken kullanılmıřtır. Dijkstra en kısa yol algoritması, `dijkstra` fonksiyonu sayesinde gerekleřtirilmektedir. Fonksiyon ierisinde Dijkstra en kısa yol algoritması, temel olarak `while (sayi<MAKS - 1)` kod bloęu ierisinde gerekleřtirilmektedir. Bu kod bloęunda rnek Őekillerde gsterildięi gibi adım adım iřlem yapılmaktadır. Her iřlem adımında en kk mesafeyi barındıran dęmler seilmekte ve ziyaretlere o dęmler ile devam edilmektedir. Kod bloęunda adım adım her dęme ulařmak iin en kısa mesafe tespit edilmektedir. Daha sonraki program kodlarında ise bu mesafeyi elde etmek iin hangi dęmlerin ziyaret edilmesi gerektięi ekrana yazdırılmaktadır. Program alıřtıęında kullanııcıdan bařlangı dęmn girmesi istenmektedir. Bařlangı dęm olarak 0'ın girildięi durumdaki program kodunun ıktısı ařaęıdaki gibidir:

```

Baslangic dugumunu sec: 0
0. dugum icin en kısa mesafe=0
Yol = 0
1. dugum icin en kısa mesafe=4
Yol = 0 1
2. dugum icin en kısa mesafe=7
Yol = 0 1 2
3. dugum icin en kısa mesafe=8
Yol = 0 3
4. dugum icin en kısa mesafe=15
Yol = 0 3 4

```

SIRA SİZDE



`dijkstra_en_kisa_yol.c` program kodlarını ařaęıdaki bilgiler doęrultusunda tekrar dzenleyip alıřtırarak uygun ıktıyı verdięini kontrol ediniz.

- `MAKS` isimli deęiřkenin boyutunu Őekil 8.9'daki toplam dęm sayısını gz nne olarak gncelleyiniz.
- `main` fonksiyonunun ierięini Őekil 8.9'daki rnek izge iin alıřacak Őekilde dzenleyiniz ve Dijkstra en kısa yol algoritmasının alıřması sonrasında ekranda nasıl sonu verdięini gzleyiniz.

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 yönsüz çizge olmak üzere ikiye ayrılırlar. Yönlü çizgelerin kenar bağlantılarında yönleri temsil eden oklar bulunurken, yönsüz çizgelerde böyle bir durum söz konusu değildir. Çizgelerde komşuluk ve çakışım olmak üzere iki farklı ilişki türü vardır. Komşuluk, iki düğüm arasında bir bağlantı olmasını ifade eder. Çakışım 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 faydalanılır. Önce derinliğine arama, çizgenin bir düğümünden başlanarak bu düğümün komşusu üzerinden gidilebilecek en uzak düğüme kadar olan noktaların ziyaret edildiği ve daha sonra geri dönülerek aynı işlemlerin ziyaret edilmemiş düğümler için sürdürüldüğü bir arama algoritmasıdır. Algoritmanın uygulanması esnasında yığın veri yapısından faydalanılır.

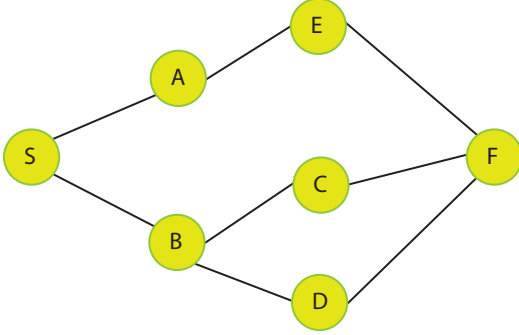


Çizgelerde en kısa yol bulma algoritmalarını kullanmak

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 varsayalım. Düğümler arasındaki ağırlıklandırılmış kenar bağlantıları ise her iki şehir çifti arasındaki uzaklığı temsil 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 varsayılır. Algoritma, düğümler arasındaki en kısa yolları aradığı için düğümler arasında daha kısa yollar bulunduğunda sonsuz ifadesi bu yolun mesafesi ile değiştirilir.

Kendimizi Sınavalım

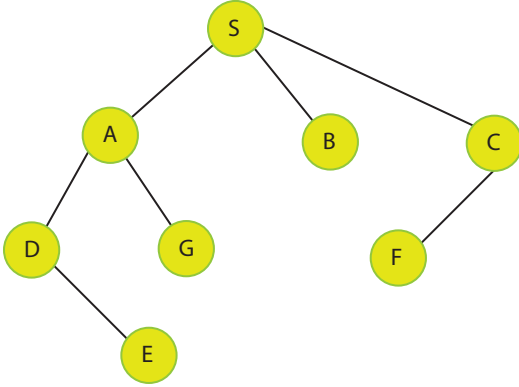
1.



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?

- S, A, E, B, C, F, D
- S, A, E, F, B, C, D
- S, A, E, B, C, D, F
- S, A, B, C, E, D, F
- S, A, B, E, C, D, F

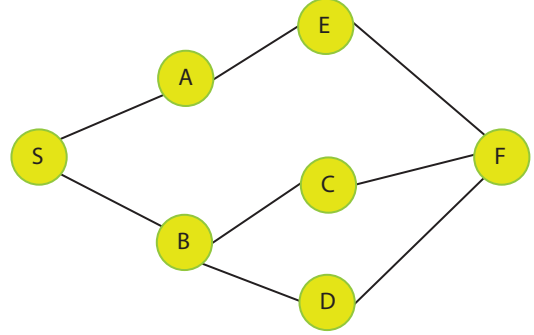
2.



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?

- S, A, B, C, D, G, F, E
- S, A, B, C, D, G, E, F
- S, A, B, C, D, F, G, E
- S, A, B, D, C, G, F, E
- S, A, B, D, C, E, F, G

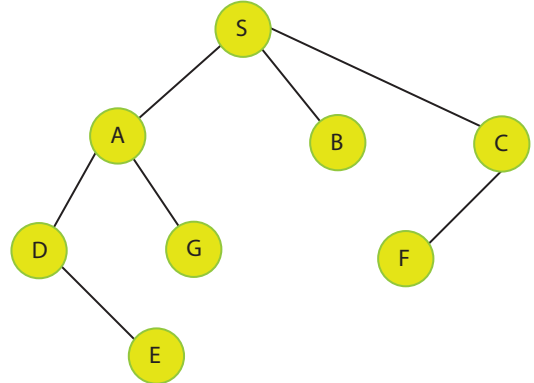
3.



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?

- S, A, E, F, B, C, D
- S, A, E, F, C, B, D
- S, A, B, C, D, E, F
- S, B, D, F, C, A, E
- S, B, C, F, D, A, E

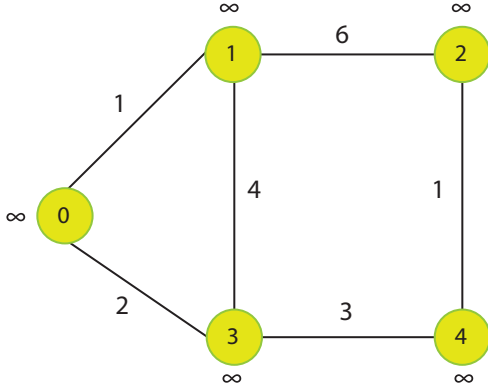
4.



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?

- S, A, D, G, B, C, F, E
- S, A, B, C, D, G, F, E
- S, B, C, F, A, G, D, E
- S, A, D, E, G, B, C, F
- S, A, D, G, E, B, C, F

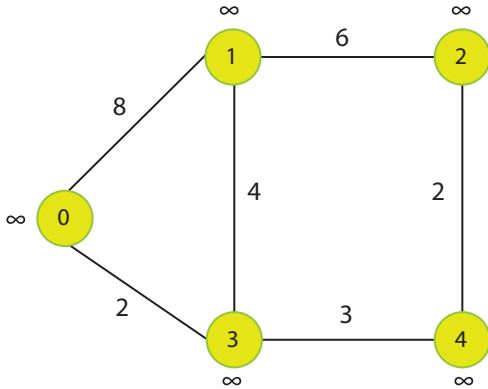
5.



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?

- 0, 1, 2, 6, 6
- 0, 1, 7, 7, 1
- 0, 1, 7, 2, 0
- 0, 1, 7, 2, 5
- 0, 1, 6, 2, 5

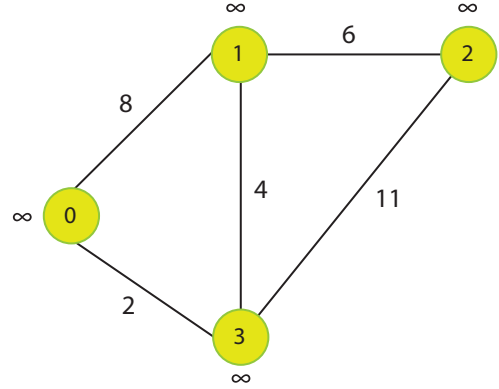
6.



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?

- 0, 6, 7, 5, 5
- 0, 3, 2, 4, 5
- 0, 7, 6, 5, 5
- 0, 6, 1, 2, 5
- 0, 6, 7, 2, 5

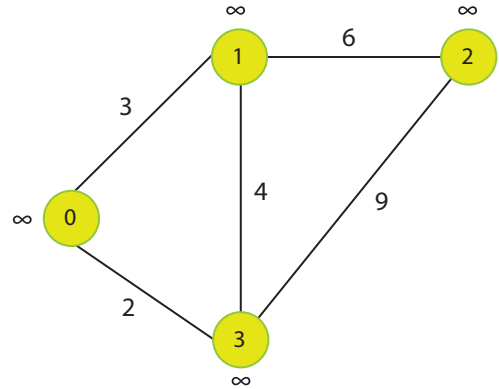
7.



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?

- 0, 3, 1, 0
- 0, 1
- 0, 3, 2
- 0, 3, 1, 2
- 0, 1, 2

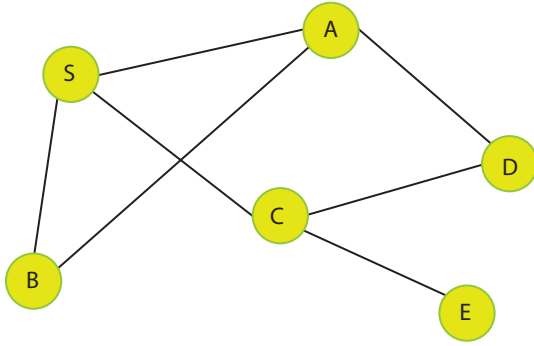
8.



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?

- 0, 1, 2
- 0, 3, 2
- 0, 3, 1, 2
- 0, 3, 1, 0, 2
- 0, 1, 3, 2

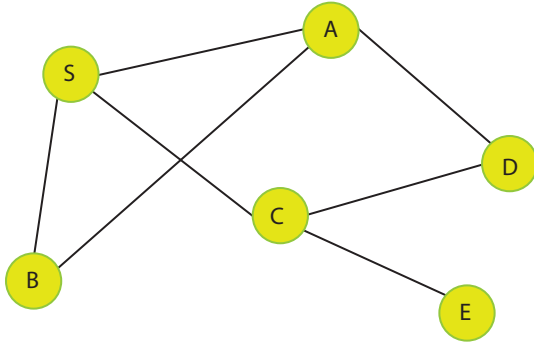
9.



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?

- S, A, D, C, E, B
- S, C, D, E, A, B
- S, A, B, C, D, E
- S, A, B, D, C, 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?

- S, A, B, C, D, E
- S, A, C, B, D, E
- S, A, B, D, C, E
- S, D, A, C, B, E
- S, C, D, E, B, A

Kendimizi Sınyalım Yanıt Anahtarı

1. e Yanıtınız yanlış ise “Enine Arama” konusunu yeniden gözden geçiriniz.
2. a Yanıtınız yanlış ise “Enine Arama” konusunu yeniden gözden geçiriniz.
3. b Yanıtınız yanlış ise “Önce Derinliğine Arama” konusunu yeniden gözden geçiriniz.
4. d Yanıtınız yanlış ise “Önce Derinliğine Arama” konusunu yeniden gözden geçiriniz.
5. e Yanıtınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
6. e Yanıtınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
7. d Yanıtınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
8. a Yanıtınız yanlış ise “Dijkstra En Kısa Yol Algoritması” konusunu yeniden gözden geçiriniz.
9. c Yanıtınız yanlış ise “Enine Arama” konusunu yeniden gözden geçiriniz.
10. c Yanıtınız 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` değışkeninin 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` fonksiyonunun 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;
    // komşuluk matrisini baslangicta sıfırla
    for (i = 0; i<MAKS; i++)
    {
        for (j = 0; j<MAKS; j++)
            komşulukmatrisi[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 algoritmasini 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.