

VER YAPILARI

Asst.Prof.Dr. HAKAN KUTUCU

DATA
STRUCTURES

HAKAN KUTUCU

VER YAPILARI

(DATA STRUCTURES)

Düzenleyen
SA M MEHMET ÖZTÜRK

KARABÜK ÜNİVERSİTESİ
Mühendislik Fakültesi Merkez Kampüsü – Karabük 2014

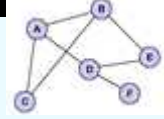


çindekiler

1. VERİ TİPLERİ	7
GİRİŞ	7
Veri Yapısı	8
Veriden Bilgiye Geçi	8
Belleğin Yapısı ve Veri Yapıları	9
Adres Operatörü ve Pointer Kullanımı	10
Yaygın Olarak Kullanılan Veri Yapıları Algoritmaları	11
2. VERİ YAPILARI	12
GİRİŞ	12
ÖZYEMEL FONKSİYONLAR	14
Rekürsif bir fonksiyonun genel yapısı	15
C'DE YAPILAR	16
Alternatif struct tanımları	17
VERİ YAPILARI	17
Matematiksel ve mantıksal modeller (<i>Soyut Veri Tipleri – Abstract Data Types - ADT</i>)	17
Uygulama (<i>Implementation</i>)	18
BAĞLI LİSTELER (<i>Linked Lists</i>)	18
Bağlı Listelerle İşlemler	18
TEK BAĞLI DÖRUSAL LİSTELER	19
Tek Bağlı Dörsal Liste Oluşturmak ve Eleman Ekleme	19
Tek Bağlı Dörsal Listenin Başına Eleman Ekleme	20
Tek Bağlı Dörsal Listenin Sonuna Eleman Ekleme	20
Tek Bağlı Dörsal Liste Elemanlarının Tüm Bilgilerini Yazdırmak	20
Tek Bağlı Dörsal Listenin Elemanlarını Yazdırmak	20
Tek Bağlı Dörsal Listenin Elemanlarını Saymak	21
Tek Bağlı Dörsal Listelerde Arama Yapmak	21
Tek Bağlı Dörsal Listelerde ki Listeyi Birleştirmek.....	22
Tek Bağlı Dörsal Listelerde Verilen Bir Dörsal Sahip Düümünü Silme	22
Tek Bağlı Dörsal Listelerde Verileri Tersten Yazdırmak	22
Tek Bağlı Dörsal Listenin Kopyasını Oluşturmak.....	23
Tek Bağlı Dörsal Listeyi Silme.....	23
main() Fonksiyonu	23
TEK BAĞLI DAİRESEL (<i>Circle Linked</i>) LİSTELER	25
Tek Bağlı Dairesel Listelerde Başına Eleman Ekleme	25
Tek Bağlı Dairesel Listelerde Sona Eleman Ekleme	25
Tek Bağlı Dairesel Listelerde ki Listeyi Birleştirmek	26
Tek Bağlı Dairesel Listelerde Arama Yapmak	26
Tek Bağlı Dairesel Listelerde Verilen Bir Dörsal Sahip Düümünü Silme	27
ÇİFT BAĞLI DÖRUSAL (<i>Double Linked</i>) LİSTELER	28
Çift Bağlı Dörsal Listenin Başına Eleman Ekleme	28
Çift Bağlı Dörsal Listenin Sonuna Eleman Ekleme	28
Çift Bağlı Dörsal Listelerde Araya Eleman Ekleme	29
Çift Bağlı Dörsal Listelerde Verilen Bir Dörsal Sahip Düümünü Silme	29
Çift Bağlı Dörsal Listelerde Arama Yapmak	30
Çift Bağlı Dörsal Listelerde Karşılaşturma Yapmak	30

Çift Ba lı Do rusal Listelerin Avantajları ve Dezavantajları	30
Ç FT BA LI DA RESEL(<i>Double Linked</i>) L STELER	31
Çift Ba lı Dairesel Listelerde Ba a Dü üm Ekleme	31
Çift Ba lı Dairesel Listenin Sonuna Eleman Ekleme	31
Çift Ba lı Dairesel Listelerde ki Listeyi Birle tirmek	32
Çift Ba lı Dairesel Listelerde Araya Eleman Ekleme	32
3.YI INLAR (<i>Stacks</i>)	33
YI INLARA (<i>Stacks</i>) G R	33
STACK'LER N D Z (<i>Array</i>) MPLEMENTASYONU	33
Stack'lere Eleman Ekleme lemi (<i>push</i>)	34
Bir Stack'in Tüm Elemanlarını Silme lemi (<i>reset</i>)	34
Stack'lerden Eleman Çıkarma lemi (<i>pop</i>)	34
STACK'LER N BA LI L STE (<i>Linked List</i>) MPLEMENTASYONU	35
Stack'in Bo Olup Olmadı ının Kontrolü (<i>isEmpty</i>)	36
Stack'in Dolu Olup Olmadı ının Kontrolü (<i>isFull</i>)	36
Stack'lere Yeni Bir Dü üm Ekleme (<i>push</i>)	36
Stack'lerden Bir Dü ümü Silme (<i>pop</i>)	37
Stack'in En Üstteki Verisini Bulma (<i>top</i>)	37
Bir Stack'e Ba langıç De erlerini Verme (<i>initialize</i>)	37
Stack'ler Bilgisayar Dünyasında Nerelerde Kullanılır	38
INFIX, PREFIX VE POSTFIX NOTASYONLARI	38
Infix notasyonu	39
Prefix notasyonu	39
Postfix notasyonu	39
4.QUEUES (Kuyruklar)	41
G R	41
KUYRUKLARIN D Z (<i>Array</i>) MPLEMENTASYONU	41
Bir Kuyru a Ba langıç De erlerini Verme (<i>initialize</i>)	43
Kuyru un Bo Olup Olmadı ının Kontrolü (<i>isEmpty</i>)	43
Kuyru un Dolu Olup Olmadı ının Kontrolü (<i>isFull</i>)	43
Kuyru a Eleman Ekleme (<i>enqueue</i>)	43
Kuyruktan Eleman Çıkarma lemi (<i>dequeue</i>)	44
KUYRUKLARIN BA LI L STE (<i>Linked List</i>) MPLEMENTASYONU	44
Kuyru a Ba langıç De erlerini Verme (<i>initialize</i>)	44
Kuyru un Bo Olup Olmadı ının Kontrolü (<i>isEmpty</i>)	4
Kuyru un Dolu Olup Olmadı ının Kontrolü (<i>isFull</i>)	45
Kuyru a Eleman Ekleme (<i>enqueue</i>)	45
Kuyruktan Eleman Çıkarma lemi (<i>dequeue</i>)	46
5.A AÇLAR (<i>Trees</i>)	51
G R	51
A AÇLARIN TEMS L	52
Tüm A açlar çin	53
kili A açlar (<i>Binary Trees</i>) çin	54
kili A açlar Üzerinde Dola ma	54
Preorder (<i>Önce Kök</i>) Dola ma	55
Inorder (<i>Kök Ortada</i>) Dola ma	55
Postorder (<i>Kök Sonda</i>) Dola ma	55
kili A aç Olu turmak	56

kili A ağca Veri Ekleme	56
K L ARAMA A AÇLARI (BSTs - <i>Binary Search Trees</i>)	59
kili Arama A ağca Veri Ekleme	59
Bir A ağca Dü ümlerinin Sayısını Bulmak	59
Bir A ağca Yüksekli ini Bulmak	60
kili Arama A ağca Bir Dü üm Silme	61
kili Arama A ağca Bir Dü ümü Bulmak	64
kili Arama A ağca Kontrolü	65
kili Arama A ağca Minimum Elemanı Bulmak	65
kili Arama A ağca Maximum Elemanı Bulmak	65
Verilen ki A ağca Kar ıla tırmak	65
Alı tırmalar	65
AVL A AÇLARI	66
Önerme	68
spat	68
Bir AVL A ağca Yapısı	70
spat	70
ddia	71
spat	71
AVL A ağca Ekleme lemi	72
Bir AVL A ağca Dü ümleri Döndürme	73
Tek Döndürme (<i>Single Rotation</i>)	73
Çift Döndürme (<i>Double Rotation</i>)	76
AVL A ağca Silme lemi	79
ÖNCEL KL KUYRUKLAR (<i>Priority Queues</i>)	81
Binary Heap (<i>kili Yı n</i>)	81
Mapping (<i>E leme</i>)	82
Heap lemleri	83
Insert (<i>Ekleme</i>)	83
Delete (<i>Silme</i>)	85
GRAPHS (<i>Çizgeler</i>)	87
G R	87
Terminoloji, Temel Tanımlar ve Kavramlar	88
GRAFLARIN BELLEK ÜZER NDE TUTULMASI	91
Kom uluk Matrisi (<i>Adjacency Matrix</i>)	91
Kom uluk Listesi (<i>Adjacency List</i>)	92
Kom uluk Matrisleri ve Kom uluk Listelerinin Avantajları-Dezavantajları	92



BÖLÜM

Veri Tipleri 1

1.1GR

Programlamada veri yapıları en önemli unsurlardan birisidir. Program kodlarını yazarken kullanılacak veri yapısının en ideal şekilde belirlenmesi, belleğin ve çalışma biçiminin daha etkin kullanılmasını sağlar. Program içerisinde işlenecek veriler diziler ile tanımlanmış bir veri bloğu içerisinde seçilebileceği gibi, işaretçiler kullanılarak daha etkin şekilde hafızada saklanabilir. Veri yapıları, dizi ve işaretçiler ile yapılmasının yanında, nesnelere de uygulanabilir.

Veri, bilgisayar ortamında sayısal, alfasayısal veya mantıksal biçimlerde ifade edilebilen her türlü değer (örneğin; 10, -2, 0 tamsayıları, 27.5, 0.0256, -65.253 gerçel sayıları, 'A', 'B' karakterleri, "Yağmur" ve "Merhaba" stringleri, 0,1 mantıksal değerleri, ses ve resim sinyalleri vb.) tanımlanarak ifade edilebilir.

Bilgi ise, verinin işlenmesi ve bir anlam ifade eden halidir. Örneğin; 10 kg, -2 derece, 0 noktası anlamlarındaki tamsayılar, 27.5 cm, 0.0256 gr, -65.253 volt anlamlarındaki gerçel sayılar, 'A' bina adı, 'B' sınıfın üyesi anlamlarındaki karakterler, "Yağmur" öğrencinin ismi, "Merhaba" selamlama kelimesi stringleri, boş anlamında 0, dolu anlamında 1 mantıksal değerleri, anlamı bilinen ses ve resim sinyalleri verilerin bilgi haline dönüşümüdür.

Veriler büyüklüklerine göre bilgisayar belleğinde farklı boyutlarda yer kaplarlar. Büyüklüklerine, kapladıkları alan boyutlarına ve tanım aralıklarına göre veriler Veri Tip'leri ile sınıflandırılmaktadır. Tablo 1.1'de ANSI/ISO Standardına göre C dilinin veri tipleri, bit olarak bellekte kapladıkları boyutları ve tanım aralıkları görülmektedir.

Tipi	Bit Boyutu	Tanım Aralığı
char	8	-127 - 127
unsigned char	8	0 - 255
signed char	8	-127 - 127
int	16 veya 32*	-32,767 - 32,767
unsigned int	16 veya 32*	0 - 65,535
signed int	16 veya 32*	-32,767 - 32,767
short int	16	-32,767 - 32,767
unsigned short int	16	0 - 65,535
signed short int	16	-32,767 - 32,767
long int	32	-2,147,483,647 - 2,147,483,647
signed long int	32	-2,147,483,647 - 2,147,483,647
unsigned long int	32	0 - 4,294,967,295
float	32	$3.4 \times 10^{-38} - 3.4 \times 10^{38}$
double	64	$1.7 \times 10^{-308} - 1.7 \times 10^{308}$

* İmceye göre 16 veya 32 bitlik olabilmektedir.

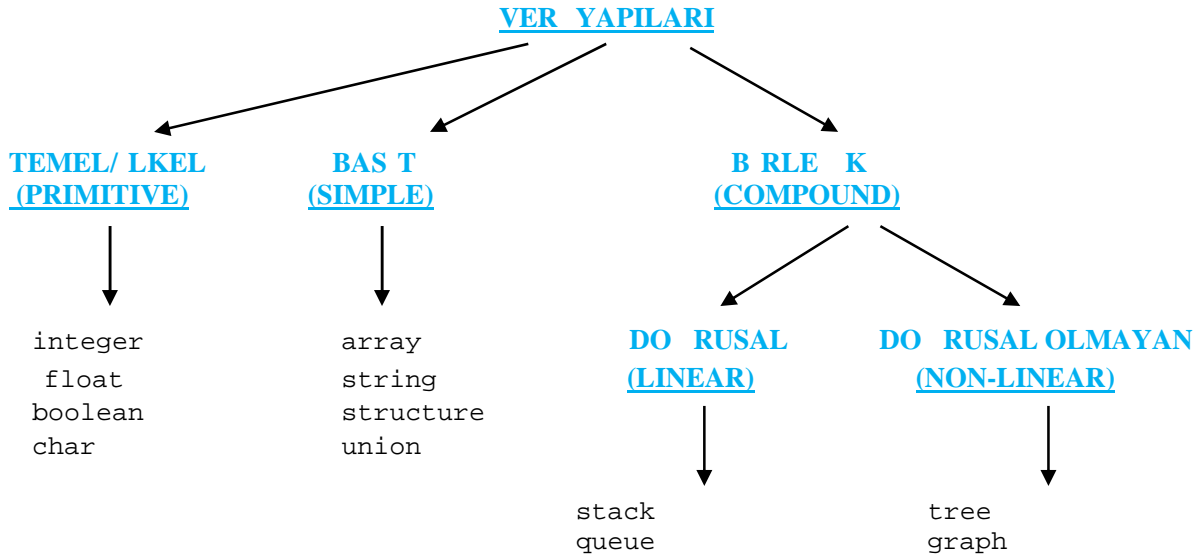
Tablo1.1 C'de veri tipleri ve tanım aralıkları.

Her programlama dilinin tablodakine benzer, kabul edilmi veri tipi tanımlamaları vardır. Programcı, programını yazacağı problemi incelerken, program algoritmasını oluştururken, programda kullanılacak değişken ve sabitlerin veri tiplerini bu tanımlamaları dikkate alarak belirler. Çünkü veriler bellekte tablodaki veri tiplerinden kendisine uygun olanlarının özelliklerinde saklanır.

Program, belleğe saklama/yazma ve okuma işlemlerini, işlemci aracılığı ile işletim sistemine yaptırır. Yani programın çalışması süresince program, işlemci ve işletim sistemi ile birlikte iletişim halinde, belleği kullanarak işi ortaya koyarlar. Veri için seçilen tip bilgisayarın birçok kısmını etkiler, ilgilendirir. Bundan dolayı uygun veri tipi seçimi programlamanın önemli bir amacıdır. Programcının doğru karar verebilmesi, veri tiplerinin yapılarını tanımasına bağlıdır. Tabloda verilen veri tipleri C programlama dilinin Temel Veri Yapılarıdır. C ve diğer dillerde, daha ileri düzeyde veri yapıları da vardır.

Veri Yapısı

Verileri tanımlayan veri tiplerinin, birbirleriyle ve hafızayla ilgili tüm teknik ve algoritmik özellikleridir. C dilinin Veri Yapıları ekil 1.1'deki gibi sınıflandırılabilir.



ekil 1.1 C Dilinin veri yapıları.

ekilden de görüleceği ve ileriki bölümlerde anlatılacağı üzere, C Veri Yapıları Temel/ İkel (*primitive*), Basit (*simple*), Birle ik (*compound*) olarak üç sınıfta incelenebilir;

- Temel veri yapıları, en çok kullanılan ve diğer veri yapılarının oluşturulmasında kullanılırlar.
- Basit veri yapıları, temel veri yapılarından faydalanılarak oluşturulan diziler (*arrays*), stringler, yapılar (*structures*) ve birle imler (*unions*)'dir.
- Birle ik veri yapıları, temel ve basit veri yapılarından faydalanılarak oluşturulan diğerlerine göre daha karmaşık veri yapılarıdır.

Program, işlemci ve işletim sistemi her veri yapısına ait verileri farklı biçim ve teknikler kullanarak, bellekte yazma ve okuma işlemleriyle uygulamalara taşırlar. Bu işlemlere Veri Yapıları Algoritmaları denir. Çeşitli veri yapıları oluşturmak ve bunları programlarda kullanmak programcıya programlama esnekliği sağlarken, bilgisayar donanım ve kaynaklarından etkin biçimde faydalanma olanakları sunar, ayrıca programın hızını ve etkinliğini artırır, maliyetini düşürür.

Veriden Bilgiye Geçi

Veriler bilgisayar belleğinde 1 ve 0'lerden oluşan bir "Bit" dizisi olarak saklanır. Bit dizisi biçimindeki verinin anlamı verinin yapısından ortaya çıkarılır. Herhangi bir verinin yapısı değiştirilerek farklı bilgiler elde edilebilir.

Örneğin; 0100 0010 0100 0001 0100 0010 0100 0001 32 bitlik veriyi ele alalım. Bu veri ASCII veri yapısına dönüştürülürse, her 8 bitlik veri grubu bir karaktere karşılık düşer;

$$\begin{array}{cccc} \underline{0100\ 0010} & \underline{0100\ 0001} & \underline{0100\ 0010} & \underline{0100\ 0001} \\ B & A & B & A \end{array}$$

Bu veri BCD (*Binary Coded Decimal*) veri yapısına dönüştürülürse, bitler 4'er bitlik gruplara ayrılır ve her grup bir haneye karşılık gelir;

$$\begin{array}{cccccccc} \underline{0100} & \underline{0010} & \underline{0100} & \underline{0001} & \underline{0100} & \underline{0010} & \underline{0100} & \underline{0001} \\ 4 & 2 & 4 & 1 & 4 & 2 & 4 & 1 \end{array}$$

Bu veri iaretsiz 16 bitlik tamsayı ise, her 16 bitlik veri bir iaretsiz tamsayıya karşılık düşer;

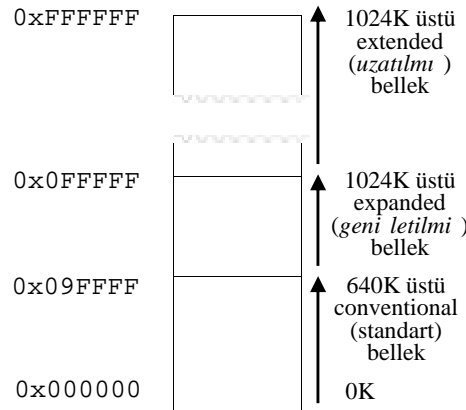
0100 0010 0100 0001 0100 0010 0100 0001
16961 16961

Bu veri i aretsiz 32 bitlik tamsayı ise, 32 bitlik bütün grup olarak bir i aretsiz tamsayıya kar ılık dü er;

0100 0010 0100 0001 0100 0010 0100 0001
1111573057

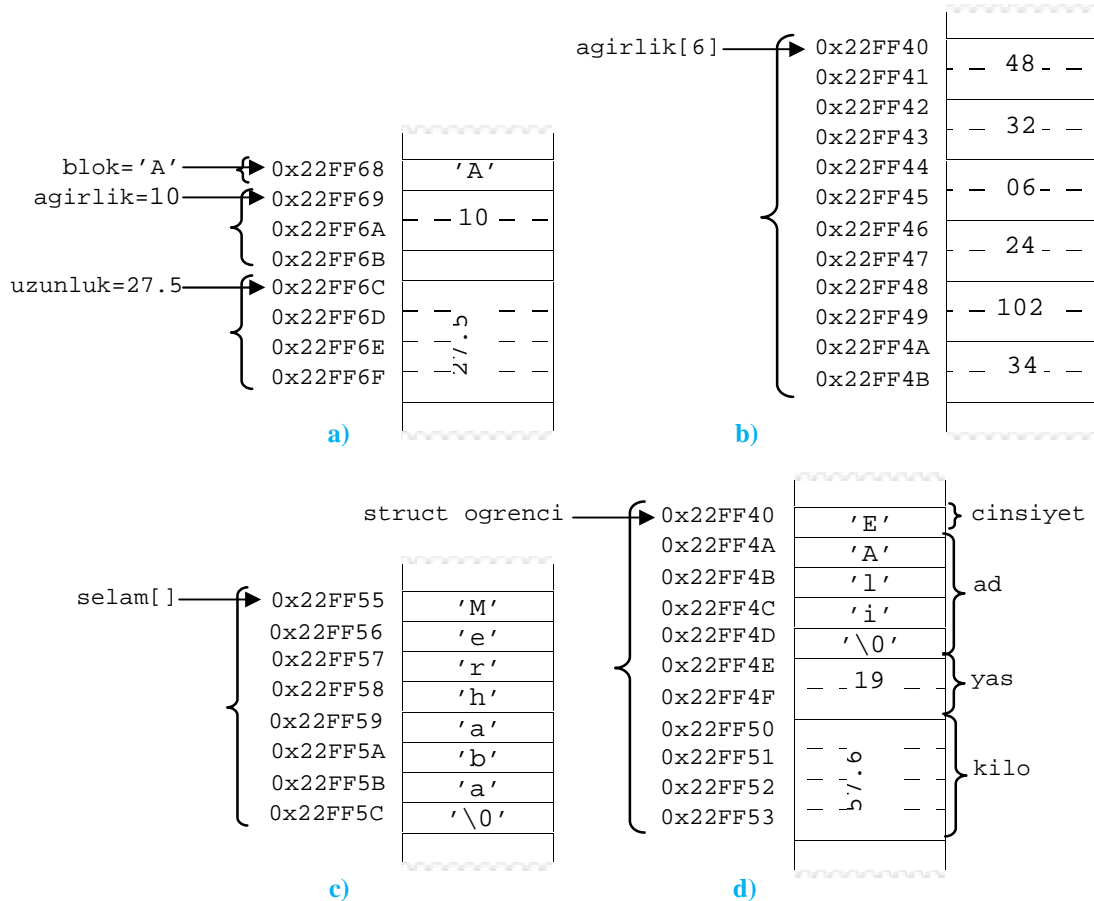
Belle in Yapısı ve Veri Yapıları

Yapısal olarak bellek, büyüklü üne ba lı olarak binlerce, milyonlarca 1'er Byte (8 Bit)'lik veriler saklayabilecek biçimde tasarlanmı bir elektronik devredir. Bellek, ekil 1.2'deki gibi her byte'ı bir hücre ile gösterilebilecek büyükçe bir tablo olarak çizilebilir.



ekil 1.2 Belle in adreslenmesi ve bellek haritası.

Her bir hücreyi birbirinden ayırmak için hücreler numaralarla adreslenir. Program, i lemci ve i letim sistemi bu adresleri kullanarak verilere eri ir (yazar/okur). Büyük olan bu adres numaraları 0'dan ba layarak bellek boyutu kadar sürer ve 16'lık (Hexadecimal) biçimde ifade edilir. letim sistemleri belle in bazı bölümlerini kendi dosya ve programlarını çalı tırmak için, bazı bölümlerini de donanımsal gereksinimler için ayırır ve kullanır. Ancak belle in büyük bölümü uygulama programlarının kullanımına ayrılmı tır. ekil 1.3'te DOS i letim sistemi için bellek haritası görülmektedir.



ekil 1.3 Veri yapılarının bellek üzerindeki yerle imleri.

Temel/ İkel (Primitive) veri yapılarından birisinin tipi ile tanımlanan bir de i ken, tanımlanan tipin özelliklerine göre bellekte yerle tirilir. Örne in;

```
char blok = 'A';
int agirlik = 10;
float uzunluk = 27.5;
```

de i ken tanımlamaları sonunda bellekte ekil 1.3 a)'daki gibi bir yerle im gerçekleşle ir. letim sistemi de i kenleri bellekteki bo alanlara veri tiplerinin özelliklerine uygun alanlarda yerle tirir.

Basit (Simple) veri yapıları temel veri yapıları ile olu turulur. Örne in;

```
int agirlik [6];
```

tanımlamasındaki agirlik dizisi 6 adet int (*tamsayı*) veri içeren bir veri yapısıdır. Bellekte ekil 1.3 b)'deki gibi bir yerle im gerçekleşle ir. letim sistemi dizinin her verisini (*elemanı*) ardı ardına, bellekteki bo alanlara veri tipinin özelliklerine uygun alanlarda yerle tirir. Örne in;

```
char selam [] = "Merhaba";
```

veya

```
char selam [] = {'M', 'e', 'r', 'h', 'a', 'b', 'a', '\0'};
```

tanımlamasındaki selam dizisi 8 adet char (*karakter*) tipinde veri içeren bir string veri yapısıdır. Bellekte ekil 1.3 c)'deki gibi bir yerle im gerçekleşle ir. letim sistemi stringin her verisini (*elemanı*) ardı ardına, bellekteki bo alanlara veri tipinin özelliklerine uygun alanlarda yerle tirir. Örne in;

```
struct kayit {
    char cinsiyet;
    char ad[];
    int yas;
    float kilo;
}ö renci;
```

tanımlamasında kayit adında bir structure (*yapı*) olu turulmu tur. Bu veri yapısı dikkat edilirse farklı temel veri yapılarından olu an, birden çok de i ken tanımlaması (*üye*) içermektedir. ogrenci, kayit yapısından bir de i kendir ve ogrenci de i keni üyelerine a a ıdaki veri atamalarını yaptıktan sonra, bellekte ekil 1.3 d)'deki gibi bir yerle im gerçekleşle ir.

```
ogrenci.cinsiyet = 'E';
ogrenci.ad[] = "Ali";
ogrenci.yas = 19;
ogrenci.kilo = 57.6;
```

letim sistemi kayit veri yapısına sahip ö renci de i keninin her bir üyesini ardı ardına ve bir bütün olarak bellekteki bo alanlara üyelerin veri tiplerinin özelliklerine uygun alanlarda yerle tirir.

Birle ik (Compound) veri yapıları basit veri yapılarından dizi veya structure tanımlamaları ile olu turulabilece i gibi, nesne yönelimli programlamanın veri yapılarından class (*sınıf*) tanımlaması ile de olu turulabilir. lerleyen bölümlerde structure ile yeni veri yapılarının tanımlama ve uygulamaları anlatılmaktadır.

Adres Operatörü ve Pointer Kullanımı

Daha önce yaptı ımız veri yapıları tanımlamalarında, verilere de i ken adları ile eri ilebilece i görölmektedir. Aynı verilere, de i kenlerinin adresleriyle de eri ilebilir. Bu eri im tekni i bazen tercih edilebilir olsa da, bazen kullanılmak zorunda kalınabilir.

A a ıdaki kodlar ile temel veri yapılarının adreslerinin kullanımları incelenmektedir. printf() fonksiyonu içerisindeki formatlama karakterlerine dikkat ediniz. Adres de erleri sistemden sisteme farklılık gösterebilir.

```
main() {
    int agirlik = 10;
    int *p;

    p = &agirlik;
    printf("%d\n", agirlik); // agirlik de işkeninin verisini yaz, 10 yazılır
    printf("%p\n", &agirlik); // agirlik de işkeninin adresini yaz, 0022FF44 yazılır
    printf("%p\n", p); // p de işkeninin verisini yaz, 0022FF44 yazılır
    printf("%d\n", *p); // p de işkenindeki adresteki veriyi yaz, 10 yazılır
    printf("%p\n", &p); // p de işkeninin adresini yaz, 0022FF40 yazılır
    return 0;
}
```

Basit veri yapılarının adreslerinin kullanımları da temel veri yapılarının kullanımlarına benzemektedir. Aşağıdaki kodlarla benzerlikler ve farklılıklar incelenmektedir.

```
main() {
    int agirlik[6] = {48, 32, 06, 24, 102, 34};
    int *p;

    p = agirlik; // DİKKAT, agirlik dizisinin adresi atanıyor

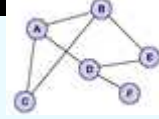
    printf("%p\n", agirlik); // agirlik dizisinin adresini yaz, 0022FF20 yazılır
    printf("%p\n", p); // p de işkeninin verisini yaz, 0022FF20 yazılır
    printf("%d\n", agirlik[0]); // Dizinin ilk elemanının verisini yaz, 48 yazılır
    printf("%d\n", *p); // p de işkeninde bulunan adresteki veriyi yaz, 48 yazılır
    printf("%d\n", agirlik[1]); // Dizinin ikinci elemanının verisini yaz, 32 yazılır
    printf("%d\n", *++p);
    // p de işkenindeki adresten bir sonraki adreste bulunan veriyi yaz, 32 yazılır

    return 0;
}
```

Dikkat edilirse tek fark üçüncü satırda p'ye agirlik dizisi doğrudan atanmıştır. Çünkü C dilinde dizi isimleri zaten dizinin başlangıç adresini tutmaktadır. Bu string'ler için de geçerlidir.

Yaygın Olarak Kullanılan Veri Yapıları Algoritmaları

- 1) Listeler,
 - a. Bir başlı doğrusal listeler,
 - b. Bir başlı dairesel listeler,
 - c. İkibaşlı doğrusal listeler,
 - d. İkibaşlı dairesel listeler,
- 2) Listeler ile stack (yığın) uygulaması,
- 3) Listeler ile queue (kuyruk) uygulaması,
- 4) Listeler ile dosyalama uygulaması,
- 5) Çok başlı listeler,
- 6) Ağaçlar,
- 7) Matrisler,
- 8) Arama algoritmaları,
- 9) Sıralama algoritmaları,
- 10) Graflar.



BÖLÜM

Veri Yapıları 2

2.1GR

Büyük bilgisayar programları yazarken karşılaştığımız en büyük sorun programın hedeflerini tayin etmektir. Hatta programı geliştirme aşamasında hangi metodları kullanacağımız hususu da bir problem değildir. Örneğin bir kurumun müdürü “*tüm demirbaşlarımızı tutacak, muhasebemizi yapacak ki isel bilgilere erişim sağlayacak ve bunlarla ilgili düzenlemeleri yapabilecek bir programımız olsun*” diyebilir. Programları yazan programcı bu ifadelerin pratikte nasıl yapıldığını tespit ederek yine benzer bir yaklaşımla programlamaya geçebilir. Fakat bu yaklaşım çoğu zaman başarısız sonuçlara gebe değildir. Programcı ifadesi yapan ahıstan aldığı bilgiye göre programa başlar ve ardından yapılan işin programa dökülmesinin çok kolay olduğunu fark eder. Lakin söz konusu bilgilerin geliştirilmekte olan programın başka bölümleri ile ilişkilendirilmesi söz konusu olunca işler biraz daha karmaşıklar. Biraz çabans, biraz da programcının ki isel mahareti ile sonuçta ortaya çalışkan bir program çıkarılabilir. Fakat bir program yazıldıktan sonra, genelde bazen küçük bazen de köklü değişikliklerin yapılmasını gerektirebilir. Esas problemler de burada başlar. Zayıf bir şekilde birbirine bağlanmış program özellikleri bu amaçta da ıllp işi göremez hale kolaylıkla gelebilir.

Bilgisayar ile ilgili işlerde en başta aklımıza bellek gelir. Bilgisayar programları gerek kendi kodlarını saklamak için veya gerekse kullandıkları verileri saklamak için genelde belleği saklama ortamı olarak seçerler. Bunlara örnek olarak karşılaştırma, arama, yeni veri ekleme, silme gibi işlemleri verebiliriz ki bunlar ayrı ayrı bellekte gerçekleştirilirler. Bu işlemlerin direkt olarak sabit disk üzerinde gerçekleştirilmesi yönünde geliştirilen algoritmalar da vardır. Yukarıda sayılan işlemler için bellekte bir yer ayrılır. Aslında bellekte her değişken için yer ayrılmıştır. Örneğin C programlama dilinde tanımladığımız bir x değişkeni için bellekte bir yer ayrılmıştır. Bu x değişkeninin adresini tutan başka bir değişken olabilir. Buna **pointer** (*i aretçi*) denir. Pointer, içinde bir değişkenin adresini tutar.

Bilgisayar belleği programlar tarafından iki türlü kullanılır:

- Statik programlama,
- Dinamik programlama.

Statik programlamada veriler programların başında sayıları ve boyutları genelde önceden belli olan unsurlardır. Örneğin;

```
int x;
double y;
int main() {
    x = 1001;
    y = 3.141;
}
```

eklinde tanımladığımız iki veri için C derleyicisi programın başlangıcından sonuna kadar tutulmuş kaydı ile bilgisayar belleğinden söz konusu verilerin boyutlarına uygun bellek yeri ayırır. Bu bellek yerleri programın yürütülmesi esnasında her seferinde x ve y değişimlerinde yapılacak olan değişiklikleri kaydederek içinde tutar. Bu bellek yerleri program boyunca statiktir. Yani programın sonuna kadar bu iki veri için tahsis edilmişlerdir ve başka bir işlem için kullanılamazlar.

Dinamik programlama esas olarak yukarıdaki çalışma mekanizmasından oldukça farklı bir durum arz eder.

```
int *x;
x = new int;
void main() {
    char *y;
    y = new char[30];
}
```

Yukarıdaki küçük programda tanımlanan x ve y i aretçi değişkenleri için `new` fonksiyonu çalıştığı zaman `int` için 4 byte'lık ve `char` için 256 byte'lık bir bellek alanını **heap** adını verdiğimiz bir nevi serbest kullanılabilir ve programların dinamik kullanımı için tahsis edilmiş olan bellek alanından ayırır. Programdan da anlaşılacağı üzere bu değişkenler

için ba langıçta herhangi bir bellek yeri ayrılması söz konusu de ildir. Bu komutlar bir döngü içerisinde yerle tirildi i zaman her seferinde söz konusu alan kadar bir bellek alanını heap'den alırlar. Dolayısıyla bir programın heap alanından ba langıçta ne kadar bellek isteminde bulunaca ı belli de ildir. Dolayısı ile programın yürütülmesi esnasında bellekten yer alınması ve geri iade edilmesi söz konusu oldu undan buradaki i lemler dinamik yer ayrılması olarak adlandırılır. Kullanılması sona eren bellek yerleri ise:

```
void free(*ptr);
```

komutu ile iade edilir. Söz konusu de i kenler ile i imiz bitti i zaman mutlaka free fonksiyonu ile bunları heap alanına geri iade etmemiz gerekir. Çünkü belli bir süre sonunda sınırlı heap bellek alanının tükenmesi ile program **out of memory** veya **out of heap** türünden bir hata verebilir.

Konuyu biraz daha detaylandırmak için bir örnek verelim; bir stok programı yaptı ımızı farz edelim ve stoktaki ürünler hakkında bazı bilgileri (*kategorisi, ürün adı, miktarı, birim fiyatı vs.*) girelim. Bu veriler tür itibarı ile tek bir de i ken ile tanımlanamazlar yani bunları sadece bir tamsayı veya real de i ken ile tanımlayamayız çünkü bu tür veriler genelde birkaç türden de i ken içeren karma ık yapı tanımlamalarını gerektirirler. Dolayısıyla biz söz konusu farklı de i ken türlerini içeren bir **struct** yapısı tanımlarız.

De i ik derleyiciler de i ik verilerin temsili için bellekte farklı boyutlarda yer ayırma yolunu seçerler. Örne in bir derleyici bir tamsayının tutulması için bellekten 2 byte'lık bir alan ayırırken, di er bir derleyici 4 byte ayırabilir. Haliyle bellekte temsil edilebilen en büyük tamsayının sınırları bu derleyiciler arasında farklılık arz edecektir.

Yukarıda sözü edilen struct tanımlaması derleyicinin tasarlanması esnasındaki tanımlamalara ba lı olarak bellekten ilgili de i kenlerin boyutlarına uygun büyüklükte bir blo u ayırma yoluna gider. Dolayısıyla stoktaki ürüne ait her bir veri giri inde bellekten bir blokluk yer isteniyor demektir. Böylece bellek, nerede bo luk varsa oradan 1 blokluk yer ayırmaktadır. Hem hızı arttırmak hem de i i kolayla tırmak için her blo un sonuna bir sonraki blo un adresini tutan bir i aretçi yerle tirilir. Daha sonra bu bellek yerine ihtiyaç kalmadı ı zaman, örne in stoktaki o mala ait bilgiler silindi inde, kullanılan hafıza alanları iade edilmektedir. Ayrıca bellek iki boyutlu de il do rusaldır. Sıra sıra hücrelere bilgi saklanır. Belle i etkin ekilde kullanmak için veri yapılarından yararlanmak gerekmektedir. Bu sayede daha hızlı ve belle i daha iyi kullanabilen programlar ortaya çıkmaktadır.

Programlama kısmına geçmeden önce bazı kavramları açıklamakta fayda vardır. Programların ço u birer function (*fonksiyon*) olarak yazılmı tır. Bu fonksiyonları yazarken dikkat edilmesi gereken nokta ise, bu fonksiyonların nasıl kullanılaca ıdır. Fonksiyonlar genellikle bir de er atanarak kullanılırlar (*parametre*). Örne in verilen nokta sayısına göre bir çokgen çizen bir fonksiyonu ele alalım. Kodu temel olarak u ekilde olmaktadır;

```
void cokgen_ciz(int kenar) {
    int i;
    ...{kodlar}
    ...
}
```

Yukarıdaki program parçasında de er olarak kenar de eri atanacaktır. Mesela be gen çizdirmek istedi imizde cokgen_ciz(5) olarak kullanmamız gerekir. Di er bir örnek ise verilen string bir ifadenin içerisindeki bo lukları altçizgi (_) ile de i tiren bir fonksiyonumuz olsun. Örne in string ifademiz "Muhendislik Fakultesi" ise fonksiyon sonucunda ifademiz "Muhendislik_Fakultesi" olacaktır. Öncelikle fonksiyonun de er olarak string türünde tanımlanmı "okul" de i kenini aldı mı ve sonucu da string olarak tanımlanmı "sonuc" de i kenine attı mı farz edelim. Fonksiyon tanımlaması u ekilde olacaktır;

```
void degistir(char *okul) {
    ...
    ...
    {fonksiyon kodları}
    ...
}
```

E er fonksiyon, bo lukları "_" ile de i tirdikten sonra yeni olu an ifadeyi tekrar okul de i kenine atasaydı fonksiyon tanımlaması u ekilde olacaktı;

```
char* degistir(char *okul) {
    ...
    {fonksiyon kodları}
    ...
    return okul;
}
```

Fark açıkça görülmektedir. Birinci programda fonksiyonun dönü tipi yok iken, ikinci programda ise char* olarak dönü tipi tanımlanmı tır. Bunun anlamı ise birinci programın okul de i keninde herhangi bir de i iklik yapmayaca ıdır.

Ancak ikinci programda `return` kodu ile okul geri döndürüldü ü için fonksiyonunun çalı tırılmasından sonra de i kenin içeri i de i ecek anlamına gelmektedir.

Bunun gibi fonksiyonlar 5 farklı türde tanımlanabilir;

- Call/Pass by Value
- Call/Pass by Reference
- Call/Pass by Name
- Call by Result
- Call by Value Result

Fonksiyon ça rısında argümanlar de ere göre ça ırma ile geçirilirse, argümanın de erinin bir kopyası olu turulur ve ça rılan fonksiyona geçirilir. Olu turulan kopyadaki de i iklikler, ça ırıcıdaki orijinal de i kenin de erini etkilemez. Bir argüman referansa göre ça rıldı nda ise ça ırıcı, ça rılan fonksiyonun de i kenin orijinal de erini ayarlamasına izin verir. ki örnekle konuyu hatırlatalım.

Örnek 2.1 Swap i lemi için call by value ve call by reference yöntemiyle fonksiyonlara ça rı yapıyor.

```
void swap_1(int x, int y) { // Call By Value
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

void swap_2(int &x, int &y) // Call By Reference
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a = 100;
    int b = 200;

    printf("Swap oncesi a'nin degeri: %d\n", a);
    printf("Swap oncesi b'nin degeri: %d\n\n", b);

    swap_1(a, b); // Call By Value

    printf("Swap_1 sonrasi a'nin degeri: %d\n", a);
    printf("Swap_1 sonrasi b'nin degeri: %d\n\n", b);

    swap_2(a, b); // Call By Reference

    printf("Swap_2 sonrasi a'nin degeri: %d\n", a);
    printf("Swap_2 sonrasi b'nin degeri: %d\n\n", b);

    getch();
    return 0;
}
```

2.2 ÖZY NELEMEL FONKS YONLAR

Özyinelemeli (*rekürsif*) fonksiyonlar kendi kendini ça ırın fonksiyonlardır. Rekürsif olmayan fonksiyonlar **iteratif** olarak adlandırılırlar. Bunların içerisinde genellikle döngüler (*for, while... gibi*) kullanılır.

Bir fonksiyon ya iteratiftir ya da özyinelemelidir. Rekürsif fonksiyonlar, çok büyük problemleri çözmek için o problemi aynı forma sahip daha alt problemlere bölerek çözmek tekni idir. Fakat her problem rekürsif bir çözüme uygun de ildir. Problemin do aşısı ona el vermeyebilir. Rekürsif bir çözüm elde etmek için gerekli olan iki adet strateji u ekildedir:

1. Kolayca çözülebilen bir temel durum (*base case*) tanımlamak, buna çıkı (*exitcase*) durumu da denir.
2. Yukarıdaki tanımı da içeren problemi aynı formda küçük alt problemlere parçalayan recursive case'dir.

Rekürsif fonksiyonlar kendilerini ça ırır. Recursive case kısmında problem daha alt parçalara bölünecek ve bölündükten sonra hatta bölünürken kendi kendini ça ıracaktır. Temel durumda ise çözüm a ikârdır.

Rekürsif bir fonksiyonun genel yapısı

Her rekürsif fonksiyon mutlaka *if* segmenti içermelidir. Eğer içermezse rekürsif durumla base durumu ayırt edilemez. Bu segmentin içerisinde de bir base-case artı olmalıdır. Eğer base-case durumu sağlanıyorsa sonuç rekürsif bir uygulama olmaksızın iteratif (*özyinelemesiz*) olarak hesaplanır. Eğer base-case artı sağlanmıyorsa else kısmında problem aynı formda daha küçük problemlere bölünür (*bazı durumlarda alt parçalara bölünemeyebilir*) ve rekürsif (*özyinelemeli*) olarak problem çözülür.

Bir fonksiyonun özyinelemeli olması, o fonksiyonun daha az maliyetli olduğunu anlamına gelmez. Bazen iteratif fonksiyonlar daha hızlı ve belleği daha az kullanarak çalışabilirler.

```
if(base case şartı)
    özyinelemesiz (iteratif olarak) hesapla
else { /* recursive case
    Aynı forma sahip alt problemlere böl
    Alt problemleri rekürsif olarak çöz
    Küçük çözümleri birleştirerek ana problemi çöz */
}
```

Örnek olarak Faktöryel (*Factorial*) problemi verilebilir.

```
4! = 4.3.2.1 = 24
4! = 4.3! // Görüldü ü gibi aynı forma sahip daha küçük probleme
3! = 3.2! // böldük. 4'ten 3'e düşürdük ve 3! şeklinde yazdık.
2! = 2.1! // Geri kalanları da aynı şekilde alt problemlere
1! = 1.0! // böldük. 0! ya da 1! base-case durumuna yazılabilir.
0! = 1
```

Yukarıdaki problemi sadece nasıl çözeriz şeklinde düşünmeyip, genel yapısını düşünmemiz gerekir. Matematiksel olarak düşünürsek problem $n(n-1)!$ şeklindedir. Yapıyı da düşünürsek,

$$n! = \begin{cases} 1 & \text{eğer } n = 0 \\ n \cdot (n-1)! & \text{eğer } n > 0 \end{cases} \text{ şeklinde olacaktır.}$$

Bu da bir temel durum içerir. Bu temel durum, eğer $n = 0$ ise sonuç 1'dir. Değilse $n \cdot (n-1)!$ olacaktır. Şimdi bunu koda dökelim;

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

Fonksiyona 4 rakamını gönderdiğimizde düşünelim. 4 sıfıra gitmediğinden fonksiyon else kısmına gidecek ve $4 * \text{fact}(3)$ ifadesiyle kendini 3 rakamıyla tekrar çağıracaktır. Bu adımları aşağıda gösterilmiştir;

<pre>4 * fact(3) ↓ 3 * fact(2) ↓ 2 * fact(1) ↓ 1 * fact(0) ↓ (1)</pre>	<p>şekilde görüldü ü gibi en alttaki fonksiyon çağırısında iteratif durum gerçekleştiği için fonksiyon sonlanacak ve ilk çağırıldığı noktaya geri dönecektir. Bu esnada bellekte $\text{fact}(0)$ çağırısı yerine 1 yazılacağı için o satırdaki durum $1 * 1$ şeklini alacak ve 1 sonucu üretilecektir. Bu sefer $\text{fact}(1)$ çağırısının yapıldığı yere dönecek ve yine $2 * 1$ şeklini alıp 2 üretecektir. Sonra $\text{fact}(3)$ çağırısının olduğu satırda $3 * 2$ şeklini alıp 6 ve $\text{fact}(4)$ çağırısının yapıldığı satıra döndükten sonra da $4 * 6$ hesaplanarak 24 sayısı üretilecektir.</p> <pre>return 4 * (return 3 * (return 2 * (return 1 * 1))) return 4 * (return 3 * (return 2 * 1)) return 4 * (return 3 * 2) return 4 * 6 return 24</pre> <p>İ biten fonksiyon bellekten silinir.</p>
--	---

Aşağıda vereceğimiz kod örneklerini siz de kendi bilgisayarınızda derleyiniz. Çıktıyı çalıştırmadan önce tahmin etmeye çalışınız. Konuyu daha iyi anlamanız için verilen örneklerdeki kodlarda yapacağımız ufak değişikliklerle farkı gözlemleyiniz.

Örnek 2.2 Rekürsif bir hesapla isimli fonksiyon tanımı yapılıyor. `main()` içerisinde de 1 rakamıyla fonksiyon çağırılıyor.

```

void hesapla(int x) {
    printf("%d", x);
    if(x < 9)
        hesapla(x + 1);
    printf("%d", x);
}
main() {
    hesapla(1);
    return 0;
}

```

Fonksiyonun çıktısına dikkat ediniz.

123456789987654321

Örnek 2.3 Klavyeden girilen n de erine kadar olan sayıların toplamını hesaplayan rekürsif fonksiyonu görüyorsunuz.

```

int sum(int n) {
    if(n == 1)
        return 1;
    else
        return n + sum(n - 1);
}

```

Örnek 2.4 Fibonacci dizisi, her sayının kendinden öncekiyle toplanması sonucu olu an bir sayı dizisidir. A a ıda klavyeden girilecek n de erine kadar olan fibonacci dizisini rekürsif olarak hesaplayan fonksiyon görölüyor. Çıktıya dikkat ediniz.

```

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

```

Örnek 2.5 Girilen 2 sayının en büyük ortak bölenini hesaplayan rekürsif fonksiyon alttaki gibi yazılabilir.

```


int ebob(int m, int n) {
    if((m % n) == 0)
        return n;
    else
        return ebob(n, m % n);
}

```

2.3 C'DE YAPILAR

struct: Birbirleri ile ilgili birçok veriyi tek bir isim altında toplamak için bir yoldur. Örne in programlama dillerinde reel sayılar için double, tamsayılar için int yapısı tanımlıyken, karma ık sayılar için böyle bir ifade yoktur. Bu yapıyı struct ile olu turmak mümkündür.

Örnek 2.6 Bir z karma ık sayısını ele alalım.

$$z = x + iy$$


real imaginary

Yapı tanımlamanın birkaç yolu vardır. İmdi bu sayıyı tanımlayacak bir yapı olu turalım ve bu yapıdan bir nesne tanımlayalım;

```

struct complex {
    int real;
    int im;
}
struct complex a, b;

```

dedi imiz zaman a ve b birer complex sayı olmu lardır. Tanımlanan a ve b'nin elemanlarına ula mak için nokta operatörü kullanırız. E er a veya b'nin birisi ya da ikisi de pointer olarak tanımlansaydı ok (->) operatörüyle elemanlarına ula acaktık. Bir örnek yapalım.

```
a.real = 4;          b.real = 6;
a.im = 7;           b.im = 9;
```

imdi de hem pointer olan hem de bir nesne olan tanımlama yapalım ve elemanlarına eri elim.

```
struct complex obj;
struct complex *p = &obj;
p -> real = 7;      obj.real = 7;
p -> im = 8;       obj.im = 8;
```

Bu örnekte complex türünde obj isimli bir nesne ve p isimli bir pointer tanımlanmı tır. p pointer'ına ise obj nesnesinin adresi atanmı tır. obj nesnesinin elemanlarına hem obj'nin kendisinden, hem de p pointer'ından eri ilebilir. p pointer'ından eri ilirken ok (->) operatörü, obj nesnesinden eri ilirken ise nokta (.) operatörü kullanıldı na dikkat ediniz. p pointer'ından eri mekle obj nesnesinde eri mek arasında hiçbir fark yoktur.

Örnek 2.7 ki karma ık sayıyı toplayan fonksiyonu yazalım.

```
struct complex {
    int real;
    int im;
}
struct complex add(struct complex a, struct complex b) {
    struct complex result;
    result.real = a.real + b.real;
    result.im = a.im + b.im;
    return result;
}
```

Alternatif struct tanımları

```
typedef struct {
    int real;
    int im;
} complex;
complex a, b;
```

Görüldü ü gibi bir typedef anahtar sözcü üyle tanımlanan struct yapısından hemen sonra yapı ismi tanımlanıyor. Artık bu tanımlamadan sonra nesnelere olu tururken ba a struct yazmak gerekmeyecektir.

```
complex a, b;
```

tanımlamasıyla complex türden a ve b isimli nesne meydana getirilmi olur.

2.4 VER YAPILARI

Veri Yapısı, bilgisayarda verinin saklanması (*tutulması*) ve organizasyonu için bir yoldur. Veri yapılarını saklarken ya da organize ederken iki ekilde çalı aca ız;

1- Matematiksel ve mantıksal modeller (Soyut Veri Tipleri – Abstract Data Types - ADT): Bir veri yapısına bakarken tepeden (*yukarıdan*) soyut olarak ele alaca ız. Yani hangi i lemlere ve özelliklere sahip oldu una bakaca ız. Örne in bir TV alıcısına soyut biçimde bakarsak elektriksel bir alet oldu u için onun açma ve kapama tu u, sinyal almak için bir anteni oldu unu görecek iz. Aynı zamanda görüntü ve ses vardır. Bu TV'ye matematiksel ve mantıksal modelleme açısından bakarsak içindeki devrelerin ne oldu u veya nasıl tasarlandıkları konusuyla ve hangi firma üretmi gibi bilgilerle ilgilenmeyece iz. Soyut bakı m içerisinde uygulama (*implementasyon*) yoktur.

Örnek olarak bir listeyi ele alabiliriz. Bu listenin özelliklerinden bazılarını a a ıda belirtirsek,

Liste;

- Herhangi bir tipte belirli sayıda elemanları saklasın,
- Elemanları listedeki koordinatlarından okusun,
- Elemanları belirli koordinattaki di er elemanlarla de i tirsin (*modifye*),
- Elemanlarda güncelleme yapsın,
- Ekleme/silme yapsın.

Verilen örnek, matematiksel ve mantıksal modelleme olarak, soyut veri tipi olarak listenin tanımıdır. Bu soyut veri tipini, yüksek seviyeli bir dilde somut hale nasıl getirebiliriz? İlk olarak **Diziler** akla gelmektedir.

2- Uygulama (Implementation): Matematiksel ve mantıksal modellemenin uygulamasıdır. Diziler, programlamada çok kullanılan veri yapıları olmasına rağmen bazı dezavantajları ve kısıtları vardır. Örneğin;

- Derleme aşamasında dizinin boyutu bilinmelidir,
- Diziler bellekte sürekli olarak yer kaplarlar. Örneğin `int` türden bir dizi tanımlandığında, eleman sayısı çarpı `int` türünün kapladığı alan kadar bellekte yer kaplayacaktır,
- Ekleme işleminde dizinin diğer elemanlarını kaydırmak gerekir. Bu işlemi yaparken dizinin boyutunun da alınması gerekir,
- Silme işlemlerinde de diğer elemanlar olacaktır.

Bu ve bunun gibi sorunların üstesinden gelmek ancak Bağlı Listelerle (*Linked List*) mümkün hale gelir.

Soyut veri tipleri (*ADT*)'nin resmi tanımını şu şekilde yapabiliriz; Soyut veri tipleri (*ADTs*) sadece veriyi ve işlemleri (*operasyonları*) tanımlar, uygulama yoktur. Bazı veri tipleri;

- Arrays (*Diziler*)
- Linked List (*Bağlı liste*)
- Stack (*Yığın*)
- Queue (*Kuyruk*)
- Tree (*Ağaç*)
- Graph (*Graf, çizge*)

Veri yapılarını çalışırken,

- 1- Mantıksal görünümüne bakacağız,
- 2- içinde barındırdığı işlemlere bakacağız (*operation*),

Bir bilgisayar programında uygulamasını yapacağız (*biz uygulamalarımızı C programlama dilini kullanarak yapacağız*).

2.5 BAĞLI LİSTELER (Linked Lists)

Liste (*list*) sözcüğü aralarında bir biçimde öncelik-sonralık ya da altlık-üstlük ilişkisi bulunan veri ögeleri arasında kurulur. Doğrusal Liste (*Linear List*) yapısı yalnızca öncelik-sonralık ilişkisini yansıtabilecek yapıdadır. Liste yapısı daha karmaşık gösterimlere imkan sağlar. Listeler temel olarak tek başlı ve çift başlı olmak üzere ikiye ayrılabilir. Ayrıca listelerin dairesel veya doğrusal olmasına göre de bir gruplandırma yapılabilir. Tek başlı listelerde *node*'lar sadece bir sonraki *node* ile bağlanarak bir liste oluştururlar. Çift başlı (*iki başlı*) listelerde ise bir *node*'da hem sonraki *node* hem de önceki *node* bağlantısı vardır. Bu bağlantılar Forward Link (*ileri başlı*) ve Backward Link (*geri başlı*) olarak adlandırılırlar. Doğrusal listelerde listede bulunan en son *node*'un başka hiçbir *node* bağlantısı yoktur. Başka olarak `NULL` alırlar. Dairesel listelerde ise en sondaki *node*, listenin başındaki *node*'a bağlanmıştır. Aşağıda buna göre yapılan sınıflandırma görülmektedir.

- Tek Başlı Listeler (*One Way Linked List*)
 - Tek Başlı Doğrusal Listeler (*One Way Linear List*)
 - Tek Başlı Dairesel Listeler (*One Way Circular List*)
- Çift Başlı listeler (*Double Linked List*)
 - Çift Başlı Doğrusal Listeler (*Double Linked Linear List*)
 - Çift Başlı Dairesel Listeler (*Double Linked Circular List*)

İlerleyen kısımlarda bu listeler ve bunlarla ilgili program parçaları anlatılacaktır. İlgililik bilinmesi gereken ayrıntı, çift başlı listelerde `previous` adında ikinci bir pointer daha vardır. Diğerleri aynı yapıdadır.

Bağlı Listeler İşlemler

Bağlı listeler üzerinde;

- 1- Liste oluşturma,
- 2- Listeye eleman eklemek,
- 3- Listedeki eleman silmek,
- 4- Arama yapmak,
- 5- Listenin elemanlarını yazmak,
- 6- Listenin elemanlarını saymak.

vb. gibi ve kusuz daha fazla işlemler yapılabilir. Bu işlemlerden bazıını açıklayalım ve fonksiyon halinde yazalım.

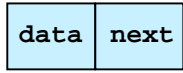
2.6 TEK BA LI DO RUSAL L STELER

Tek ba lı do rusal liste, ö elerinin arasındaki ili ki (*Logical Connection*)'ye göre bir sonraki ö enin bellekte yerle ti i yerin (*Memory Location*) bir gösterge ile gösterildi i yapıdır. Bilgisayar belle i do rusaldır. Bilgiler sıra sıra hücelere saklanır. Her bir bilgiye daha kolay ula mak için bunlara numara verilir ve her birine **node** adı verilir. Data alanı, numarası verilen node'da tutulacak bilgiyi ifade eder. Next (*link*) alanı ise bir node'dan sonra hangi node gelecekse o node'un bellekteki adresi tutulur.

Tek ba lı listelerin genel yapısı a a ıda verilmi tir. Konu anlatılırken daima bu temel yapı kullanılaca ından unutmamalıyız.

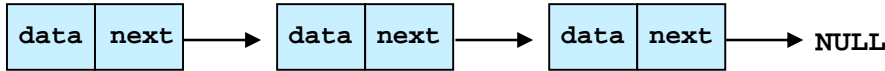
```
struct node {
    int data;
    struct node *next;
};
```

Ba lı listeler ierisindeki dü ümlerin yukarıdaki tanımlamayla iki ö esinin oldu u görülüyor. Birinci ö e olan data, her türden veri ierebilir, örne in telefon numaraları, TC kimlik numaraları vb. gibi. Biz int türden bir nesneyi yeterli bulduk. İkinci ö e olan next, bir ba lı listede mutlaka bulunması gereken bir ö edir. Dikkat edilirse struct node tipinde bir i aretçidir.



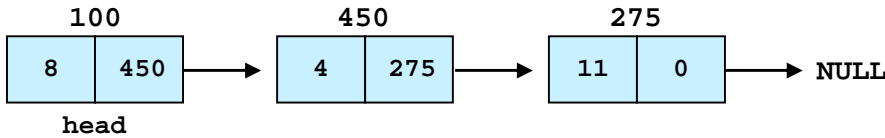
ekil 2.1 Tek Ba lı Listelerde bir node'un mantıksal yapısı.

Tek ba lı listelerin yapısında bulunan node tipindeki *next i aretçisi, rekürsif fonksiyonlarla karı tırılmamalıdır. Burada next, bir sonraki node türden dü ümün adresini gösterecektir.



ekil 2.2 Tek ba lı listeler.

Altta ekilde her çiftli kutucuk liste yapısını temsil etmektedir. Bu kutucukların üstündeki numaralar ise bellekte buldukları yerin adresidir. Burada önemli olan, next göstericisinin de erlerinin, yani dü üm adreslerinin sıralı olmayı ıdır. İlk dü ümün adresi 100, ikincisinin 450 ve üçüncüsünün ise 275'tir. Yani bellekte neresi bo sa o adresi almı lardır. Oysa dizilerde tüm elemanlar sıralı bir ekilde bellekte yer kaplıyordu.



ekil 2.3 Liste yapısı ve bellekteki adreslerinin mantıksal gösterimi.

Listenin ilk elemanı genellikle head olarak adlandırılır. head'den sonra di er elemanlara eri mek kolaydır. Bazı kaynaklarda listenin sonundaki elemanın ismi tail olarak adlandırılmı tir. Fakat biz bu ismi notlarımızda kullanmayaca ız.

Tek Ba lı Do rusal Liste Olu turmak ve Eleman Ekleme

Bu örnekte ilk olarak listeyi olu turaca ız, ardından eleman ekleme yapaca ız.

```
main() {
    struct node *head; // henüz bellekte yer kaplamıyor
    head = (struct node *)malloc(sizeof(struct node));
    // artık bellekte yer tahsis edilmiştir.

    head -> data = 1;
    head -> next = NULL;

    /* listeye yeni eleman ekleme */
    /* C++'ta head -> next = new node() şeklinde kullanılabilir. */
    head -> next = (struct node *)malloc(sizeof(struct node));
    head -> next -> data = 3;
    head -> next -> next = NULL;
```

```
}
```

Peki, eleman eklemek istersek sürekli olarak head->next->next... diye uzayacak mı? Tabii ki hayır! Elbette ki bunu yapmanın daha kolay bir yolu var.

Tek Ba lı Do rusal Listenin Ba ına Eleman Eklemek

Bir fonksiyonla örnek verelim. Siz isterseniz typedef anahtar sözcü ünü kullanarak sürekli struct yazmaktan kurtulabilirsiniz fakat biz akılda kalıcı olması açısından struct'ı kullanmaya devam edece iz.

```
// Tek ba lı do rusal listenin başına eleman eklemek
struct node *addhead(struct node *head,int key) {

    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp -> data = key;
    temp -> next = head; // temp'in next'i şu anda head'i gösteriyor.
    /* Bazen önce listenin boş olup olmadığı kontrol edilir, ama gerekli de il
       aslında. Çünkü boş ise zaten head=NULL olacaktır ve temp olan ilk dü ümün
       next'i NULL gösterecektir. */
    head = temp; /* head artık temp'in adresini tutuyor yani eklenen dü üm
                  listenin başı oldu. */

    return head;
}
```

Tek Ba lı Do rusal Listenin Sonuna Eleman Eklemek

Listenin sonuna ekleme yapabilmek için liste sonunu bilmemiz gerekiyor. Listede eleman oldu unu varsayıyoruz. Liste sonunu bulabilmek içinse bu liste elemanları üzerinde tek tek ilerlemek gerekti inden head'in adresini kaybetmemek için bir de i ken olu turaca ız ve head'in adresini bu de i kene atayaca ız.

```
struct node *addlast(struct node *head,int key) {
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    /* C++'ta struct node *temp = new node();
       * şeklinde kullanılabilen ini unutmayınız. */
    temp -> data = key;
    temp -> next = NULL;
    struct node *temp2 = head;
    /* Aşa ıdaki while yapısı traversal(dolaşma) olarak adlandırılır */
    while(temp2 -> next != NULL)
        temp2 = temp2 -> next;
    temp2 -> next = temp;
    return head;
}
```

Tek Ba lı Do rusal Liste Elemanlarının Tüm Bilgilerini Yazdırmak

Listedeki elemanların adreslerini, data kısımlarını ve sonraki dü üm adresini ekrana basan listinfo isimdeki fonksiyon aşağıdaki gibi yazılabilir.

```
void listinfo(struct node* head) {
    int i = 1;
    while(head != NULL) {
        printf("%d. Dugumunun Adresi = %p \n", i, head);
        printf("%d. Dugumunun verisi = %d \n", i, head -> data);
        printf("%d. Dugumunun Bagli Oldugu Dugumun Adresi= %p\n\n",i, head->next);
        head = head -> next;
        i++;
    }
}
```

Tek Ba lı Do rusal Listenin Elemanlarını Yazdırmak

Fonksiyon sadece ekrana yazdırma i i yapaca ından void olarak tanımlanmalıdır. Liste bo sa ekrana listede eleman olmadığı na dair mesaj da verilmelidir.

```
void print(struct node *head) {
    if(head == NULL) {
```

```

        printf("Listede eleman yok");
        return;
    }
    struct node *temp2 = head;
    while(temp2!= NULL) { // temp2->next!=NULL koşulu olsaydı son dü ü m yazılmazdı
        printf("%d\n", temp2 -> data);
        temp2 = temp2 -> next;
    }
}

```

ü phesiz elemanları yazdırmak için ö zyinelemeli bir fonksiyon da kullanılabilirdi.

```

//Tek ba lı liste elemanlarını ö zyinelemeli yazdırmak
void print_recursive(struct node *head) {
    if(head == NULL)
        return;
    printf("%d\t", head -> data);
    print_recursive (head -> next);
}

```

SORU: Yukarıdaki fonksiyon a a ıdaki gibi yazılırsa çıktısı ne olur.

```

void print_recursive2(struct node *head) {
    if(head == NULL)
        return;
    print_recursive2 (head -> next);
    printf("%d\t", head -> data);
}

```

Tek Ba lı Do rusal Listenin Elemanlarını Saymak

Listenin elemanlarını saymak için int tipinden bir fonksiyon olu turaca ız. Ayrıca listede eleman olup olmadı mı da kontrol etmeye gerek yoktur. Çünkü eleman yok ise while dö ngüsü hiç çalı mayacak ve 0 de erini dö ndürecektir.

```

int count(struct node *head) {
    int counter = 0;
    while(head != NULL) { // head->next!=NULL koşulu olsaydı son dü ü m sayılmazdı
        counter++;
        head = head -> next;
    }
    return counter;
}

```

Bu i lemi ö zyinelemeli yapmak istersek:

```

int count_recursive(struct node *head) {
    if (head == NULL)
        return 0;
    return count_recursive(head->next) + 1;
}

```

Tek Ba lı Do rusal Listelerde Arama Yapmak

Bu fonksiyon ile liste içinde arama yapılmaktadır. E er aranan bilgi varsa, bulundu u node'un adresiyle geri dö ner. Bu fonksiyon bulundu ise true bulunmadı ise false dö ndürecek ekilde de düzenlenebilir.

```

struct node* locate(struct node* head, int key) {
    struct node* locate = NULL;
    while(head != NULL)
        if(head -> data != key)
            head = head -> next;
        else {
            locate = head;
            break;
        }
    return(locate);
}

```

Tek Ba lı Do rusal Listelerde ki Listeyi Birle tirmek

list_1 ve list_2 adındaki iki listeyi birle tirmek için concatenate fonksiyonunu kullanabiliriz.

```
void concatenate(struct node*& list_1, node* list_2) { // parametrelere dikkat
    if(list_1 == NULL)
        list_1 = list_2;
    else
        last(list_1) -> next = list_2; // last isimli fonksiyona ça rı yapılıyor
}
```

Tek Ba lı Do rusal Listelerde Verilen Bir De ere Sahip Dü üümü Silmek



ekil 2.4 Tek ba lı listelerin mantıksal yapısı.

Tek ba lı listelerde verilen bir dü üümü silme i lemi, o dü ümün ba ta ya da ortada olmasına göre farklılık gösterir. İlk dü üm silinmek istenirse ikinci elemanın adresini yani head'in next i aretçisinin tuttu u adresi head'e atayarak ba taki eleman silinebilir. E er ortadan bir dü üm silinmek istenirse bir önceki dü ümü, silinmek istenen dü ümden bir sonraki dü üme ba lamak gerekir. İmdi _remove ismini verece imiz bu fonksiyonu yazalım.

```
struct node *remove(struct node *head, int key) {
    if(head == NULL) {
        printf("Listede eleman yok\n");
        return;
    }
    struct node *temp = head;
    if(head -> data == key) { // ilk dü üm silinecek mi diye kontrol ediliyor.
        head = head -> next; // head artık bir sonraki eleman.
        free(temp);
    }
    else if(temp -> next == NULL) { // Listede tek dü üm bulunabilir.
        printf("Silmek istediginiz veri bulunmamaktadır.\n\n");
        return head;
    }
    else {
        while(temp -> next -> data != key) {
            if(temp -> next -> next == NULL) {
                printf("Silmek istediginiz veri bulunmamaktadır.\n\n");
                return head;
            }
            temp = temp -> next;
        }
        struct node *temp2 = temp -> next;
        temp -> next = temp -> next -> next;
        free(temp2);
    }
    return head;
}
```

Tek Ba lı Do rusal Listelerde Verileri Tersten Yazdırmak

Tek ba lı listenin elemanlarını tersten yazdıran print_reverse adlı bir fonksiyon yazalım. Bu fonksiyonu yazarken her veriyi yeni bir listenin ba na ekleyece iz ve böylece ilk listenin tersini elde etmi olaca ız. Bunun için addhead adlı fonksiyonu kullanaca ız.. print_reverse fonksiyonunda struct node* türden yeni bir de i ken daha tanımlayaca ız ve head'in elemanlarını bu yeni listenin sürekli ba na ekleyerek verileri ters bir biçimde sıralayaca ız ve yazdırma i lemini gerçekle tirece iz. Aslında tersten yazdırma i ini rekürsif olarak yapan bir fonksiyon daha önce SORU olarak sorulmu fonksiyondur. Rekürsif fonksiyonları iyice kavramanız için bolca örnek yapmalısınız. Çünkü veri yapılarında rekürsif fonksiyonların çok büyük bir önemi vardır.

```
void print_reverse(struct node *head) {
    struct node *head2 = NULL; // yeni listenin başını tutacak adres de işkeni
```



```

struct node *temp = head;
while(temp != NULL) {
    head2 = addhead(head2, temp -> data);
    temp = temp -> next;
}
print(head2);
}

```

Tek Başlı Dorsal Listenin Kopyasını Oluşturmak

head'in kopyası oluşturulup kopya geri gönderilmektedir. kopya listesinde veriler aynı fakat adresler farklıdır.

```

struct node* copy(struct node* head) {
    struct node* kopya = NULL;
    if(head != NULL)
        do {
            concatenate(kopya, cons(head -> data));
            head = head -> next;
        } while(head != NULL);
    return kopya;
}

```

Listeyi Silmek

Listelerin kullanımı bittiğinde bunların bellekte yerini al etmemesi için tüm düğümlerinin silinmesi gereklidir. head düğümünün silinmesi listenin kullanılmaz hale gelmesine neden olur ancak head ten sonraki düğümler hala bellekte yer kaplayama devam eder.

```

struct node *destroy(struct node *head) {
    if(head == NULL) {
        printf("Liste zaten bos\n");
        return;
    }
    struct node *temp2;
    while(head!= NULL) { // while içindeki koşul temp2 -> next, NULL de ilse
        temp2=head;
        head = head->next;
        free(temp2);
    }
    return head;
}

```

SORU: destroy fonksiyonunu öz yinelemeli olarak yazınız.

Main Fonksiyonu içinde Tanımladığımız Tüm Fonksiyonların Çalıştırılması

Yukarıda tanımladığımız fonksiyonların çalıştırılması için tüm fonksiyonlar, struct tanımlaması dahil a a ıdaki main() fonksiyonu üzerinde yazılır.

```

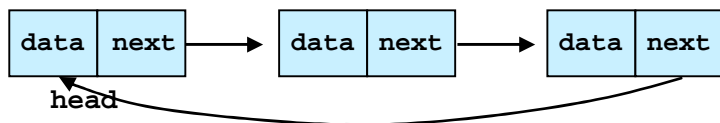
main(){
    int secim,data;
    struct node *head = NULL;
    while(1){
        printf("1-Listenin Basina Eleman Ekle\n");
        printf("2-Listenin Sonuna Eleman Ekle\n");
        printf("3-Listeyi Gorme\n");
        printf("4-Listeden verilen bir degere sahip dugum silmek\n");
        printf("5-Listeyi sil\n");
    }
}

```

```
printf("6-Listedeki eleman sayisi\n");
printf("7-Listenin tum eleman bilgileri\n");
printf("8-Programdan Cikma\n");
printf("Seciminiz....?");
scanf("%d",&secim);
switch(secim){
case 1:
    printf("Eklemek istediginiz degerini giriniz..?");
    scanf("%d",&data);
    head=addhead(head,data);
    break;
case 2:
    printf("Eklemek istediginiz degerini giriniz..?");
    scanf("%d",&data);
    head=addlast(head,data);
    break;
case 3:
    print(head);
    break;
case 4:
    printf("Silmek istediginiz degerini giriniz..?");
    scanf("%d",&data);
    head=delete(head,data);
    break;
case 5:
    head=destroy(head);
    break;
case 6:
    printf("Listede %d eleman vardir\n",count(head));
    break;
case 7:
    listinfo(head);
    break;
case 8:
    exit(1);
    break;
default: printf("Yanlis secim\n");
}
}
```

2.7 TEK BA LI DA RESEL (Circle Linked) L STELER

Tek ba lı dairesel listelerde, do rusal listelerdeki birçok i lem aynı mantık ile benzer ekilde uygulanır; fakat burada dikkat edilmesi gereken nokta, dairesel ba lı listede son elemanının next i aretçisi head'i göstermektedir..



ekil 2.5 Dairesel ba lı listeler.

Tek Ba lı Dairesel Listelerde Ba a Eleman Eklemek

Tek ba lı listelerde yapt ımızdan farklı olarak head'in global olarak tanımland ı varsayılm ıdır. Liste yoksa olu turuluyor, e er var ise, struct node* türden temp ve last dü ümleri olu turularak last'a head'in adresi atanıyor (*head'in adresini kaybetmememiz gerekiyor*). temp'in data'sına parametre de i keninden gelen veri aktarıldıktan sonra last dö ngü içerisinde ilerletilerek listenin son elemanı göstermesi sa lanıyor. temp head'i gösterecek ekilde atama yapıldıktan sonra listenin son elemanını gösteren last'ın next i aretçisine de temp'in adresi atanıyor. u anda last eklenen verinin bulundu u dü ümü gösteriyor de il mi? temp'in next i aretçisi ise head'i gösteriyor. head'e temp atanarak i lem tamamlanm ı oluyor. **D KKAT!** Artık temp'in next göstericisi, head'in bir önceki adres bilgisini tutuyor.

```

void insertAtFront(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        struct node *last = head;

        temp -> data = key;
        while(last -> next != head) // listenin son elemanı bulunuyor.
            last = last -> next;
        temp -> next = head;
        last -> next = temp;
        head = temp;
    }
}

```

Tek Ba lı Dairesel Listelerde Sona Eleman Eklemek

Fonksiyon, ba a ekleme fonksiyonuna çok benzemektedir. Listenin NULL olup olmad ı kontrol ediliyor.

```

void insertAtLast(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        struct node *last = head;
        temp -> data = key;
        // listenin son elemanı bulunuyor.
        while(last -> next != head)
            last = last -> next;
        temp -> next = head;
        last -> next = temp;
    }
}

```

Görüldü ü gibi ba a ekleme fonksiyonunun sonundaki `head = temp;` satırını kaldırmak yeterlidir. Aynı fonksiyonu a a ıdaki ekilde de yazabilirdik. Burada `temp` isminde bir de i kene ihtiyacımızın olmadı mı gözlemleyin.

```
void insertAtLast(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
    }
    else {
        struct node *last = head;
        while(last -> next != head) // listenin son elemanı bulunuyor.
            last = last -> next;
        last -> next = (struct node *)malloc(sizeof(struct node));
        last -> next -> next = head;
        last -> next -> data = key;
    }
}
```

Yazılan fonksiyonları siz de bilgisayarınızda kodlayarak mantı ı kavramaya çalı nız. Ancak çok miktarda alı tırma yaptıktan sonra iyi bir pratik kazanabilirsiniz.

Tek Ba lı Dairesel Listelerde ki Listeyi Birle tirmek

`concatenate` fonksiyonu tek ba lı dairesele listelerde iki listeyi verilen ilk listenin sonuna ekleyerek birle tiren fonksiyon olarak tanımlanacaktır. Bu fonksiyonun yazımında önemli olan do ru i lem sırasını takip etmektir. E er öncelikli yapılması gereken ba lantılar sonraya bırakılırsa son `node`'un bulunmasında sorunlar çıkacaktır. `list_1` bo ise `list_2`'ye e itleniyor. E er bo de ilse her iki listenin de `last()` fonksiyonuyla son elemanları bulunuyor ve `next` i aretçilerine bir di erinin gösterdi i adres de eri atanıyor.

```
// list_1 listesinin sonuna list_2 listesini eklemek
void concatenate(struct node*& list_1, struct node* list_2){ //parametrelere dikkat
    if(list_1 == NULL)
        list_1 = list_2;
    else {
        // Birinci listenin son dü ümünü last olarak bulmak için
        struct node *last=list_1;
        while(last -> next != list_1)
            last = last -> next;
        last->next=list_2; //Birinci listenin sonu ikinci listenin başına ba landı
        // İkinci listenin son dü ümünü last olarak bulmak için
        last=list_2;
        while(last -> next != list_2)
            last = last -> next;
        last->next=list_2; //İkinci listenin sonu birinci listenin başına ba landı
    }
}
```

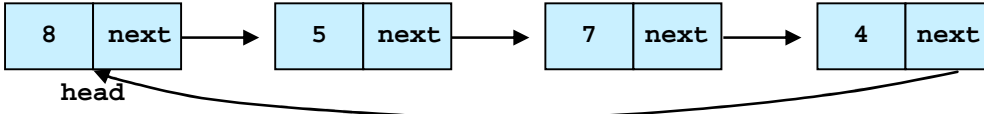
Tek Ba lı Dairesel Listede Arama Yapmak

Bu fonksiyon ile liste içinde arama yapılmaktadır. `node`'lar taranarak `data`'lara bakılır. E er aranan bilgi varsa, bulundu u `node`'un adresiyle geri döner.

```
//head listesinde data'sı veri olan node varsa adresini alma
struct node* locate(int veri, struct node* head) {
    struct node* locate = NULL;
    struct node* temp = head;
    do {
        if(head -> data != veri)
            head = head -> next;
        else {
            locate = head;
            break;
        } while(head != temp);
    } while(head != temp);
    return(locate);
}
```

Tek Ba lı Dairesel Listelerde Verilen Bir De ere Sahip Dü üümü Silmek

Silme i lemine dair fonksiyonumuzu yazmadan önce tek ba lı dairesel listelerin mantıksal yapısını tekrar gözden geçirmekte fayda vardır. Bu liste yapısında son dü üümün next i aretçisi head'i gösteriyor ve dairesel yapı sa lanıyor.



ekil 2.6 Tek ba lı listelerin mantıksal yapısı.

Tek ba lı dairesel listelerde verilen bir de ere sahip dü üümü silme i lemi için `deletenode` isimli bir fonksiyon yazacağız. Fonksiyonda liste bo ise hemen fonksiyondan çıkılır ve geriye `false` de eri döndürülür. E er silinecek dü üüm ilk dü üüm ise daha önce yapt ımız gibi ilk dü üüm listeden çıkarılır. Ancak burada listenin son dü üümünü yeni head dü üümüne ba lamak gereklidir. Bu yüzden önce son dü üüm bulunur.

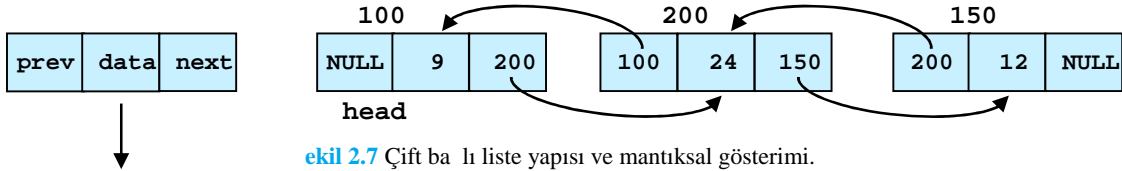
```

struct node *deletenode(struct node *head, int key) {

    if(head == NULL) {
        printf("Listede eleman yok\n");
        return;
    }
    struct node *temp = head;
    if(head -> data == key) { // ilk dü üüm silinecek mi diye kontrol ediliyor.
        struct node *last=head;
        while(last -> next != head)
            last = last -> next;
        head = head -> next; // head artık bir sonraki eleman.
        last->next=head;
        free(temp);
    }
    else if(temp -> next == NULL) { // Listede tek dü üüm bulunabilir.
        printf("Silme istediginiz veri bulunmamaktadır.\n\n");
    }
    else {
        while(temp -> next -> data != key) {
            if(temp -> next -> next == NULL) {
                printf("Silme istediginiz veri bulunmamaktadır.\n\n");
                return head;
            }
            temp = temp -> next;
        }
        struct node *temp2 = temp -> next;
        temp -> next = temp -> next -> next;
        free(temp2);
    }
    return head;
}
  
```

2.8 ÇİFT BAĞLI DOĞRUSAL (Double Linked) LİSTELER

Çift bağılı listelerin mantıksal yapısı Şekil 2.7'de gösterilmiştir. Her düğümün data adında verileri tutacağı bir de iki kenar ile kendinden önceki ve sonraki düğümlerin adreslerini tutacak olan prev ve next isiminde iki adet işaretçi vardır. Listenin başını gösteren işaretçi head yapıdadır. Şekilde head'ın adresi 100'dür ve head'ın prev işaretçisi herhangi bir yeri göstermediğinden NULL değer içermektedir. next işaretçisi ise bir sonraki düğümün adresi olan 200 de erini içermektedir. İkinci düğümün prev işaretçisi head'ın adresi olan 100 de erini tutmakta, next işaretçisi ise son düğümün adresi olan 150 de erini tutmaktadır. Nihayet son düğümün prev işaretçisi kendinden önceki düğümün adresini yani 200 de erini tutmakta ve next işaretçisi ise NULL de er içermektedir.



Şekil 2.7 Çift bağılı liste yapısı ve mantıksal gösterimi.

Çift bağılı listelerin struct yapısı aşağıda verilmiştir:

```
struct node {
    int data;
    struct node* next;
    struct node* prev;
}
```

Çift Bağılı Doğrusal Listenin Başına Eleman Ekleme

head de işaretçinin global olarak tanımlandığını varsayarak insertAtFirst fonksiyonunu yazalım.

```
void insertAtFirst(int key) {
    if(head == NULL) { // liste yoksa oluşturuluyor
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = NULL;
        head -> prev = NULL;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> data = key;
        temp -> next = head;
        temp -> prev = NULL;
        head -> prev = temp;
        head = temp;
    }
}
```

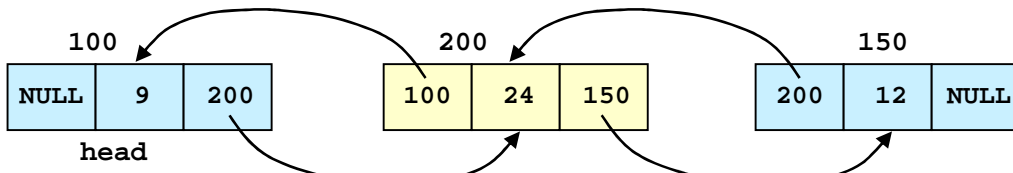
Çift Bağılı Doğrusal Listenin Sonuna Eleman Ekleme

```
void insertAtEnd(int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = NULL;
        head -> prev = NULL;
    }
    else {
        struct node *temp = head;
        struct node *temp2 = (struct node *)malloc(sizeof(struct node));
        while(temp -> next != NULL) // listenin sonunu bulmamız gerekiyor.
            temp = temp -> next;
        temp2 -> data = key;
        temp2 -> next = NULL;
        temp2 -> prev = temp;
        temp -> next = temp2;
    }
}
```

Çift Ba lı Do rusal Listelerde Araya Eleman Ekleme

```
// head listesinin n. dü ümünün hemen ardına other_node dü ümünü ekleme
void addthen(node* other_node, node*& list, int n) {
    node* temp = head;
    int i = 1;
    while(i < n) {
        head = head -> next;
        i++;
    }
    other_node -> prev = head;
    other_node -> next = head -> next;
    head -> next = other_node;
    head = temp;
}
```

Çift Ba lı Do rusal Listelerde Verilen Bir De ere Sahip Dü ümü Silmek

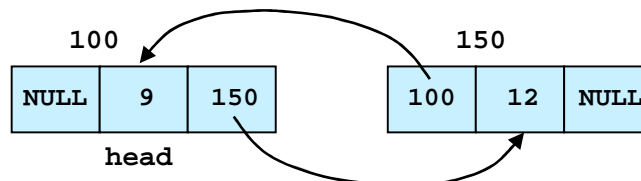


ekil 2.8 Silinmek istenen ortadaki dü üm sarı renkte gösterilmiştir.

Silme işlemi diğer liste türlerine göre biraz farklılık göstermektedir. İlk dü üm silinmek istenirse head'in next işaretçisinin tuttu u adresi head'e atadıktan sonra prev işaretçisinin de erini NULL yapmamız gerekir. Sonra free() fonksiyonuyla ba tiki eleman silinebilir. Eğer ortadan bir dü üm silinmek istenirse, silinecek dü ümün üzerinde durup bir önceki dü ümü, silinmek istenen dü ümden bir sonraki dü üme ba lamak gerekir. Silinecek dü üm ekil 2.8'de görüldü ü gibi ortadaki dü üm olsun. double_linked_remove isimli fonksiyonumuzu yazarak konuyu kavrayalım.

```
void double_linked_remove(int key) {
    struct node *temp = head;
    if(head -> data == key) { // silinecek de erin ilk dü ümde olması durumu.
        head = head -> next;
        head -> prev = NULL;
        free(temp);
    }
    else {
        while(temp -> data != key)
            temp = temp -> next;
        temp -> prev -> next = temp -> next;
        /* silinecek dü ümden bir önceki dü ümün next işaretçisi, şimdi silinecek
           dü ümden bir sonraki dü ümü gösteriyor. */
        if(temp -> next != NULL) // silinecek dü üm son dü üm de ilse
            temp -> next -> prev = temp -> prev;
        /* silinecek dü ümden bir sonraki dü ümün prev işaretçisi, şimdi
           silinecek dü ümden bir önceki dü ümü gösteriyor. */
        free(temp);
    }
}
```

Listenin, ortada bulunan dü ümü silindikten sonraki görünümünü ekil 2.9'da görüyorsunuz.



ekil 2.9 Silme işleminden sonra yeni listenin görünümü.

Çift Ba lı Do rusal Listelerde Arama Yapmak

```
// head listesinde data'sı veri olan node varsa adresini alma
struct node* locate(int veri, struct node* head) {
    struct node* locate = NULL;
    while(head != NULL) {
        if(head -> data != veri) {
            head = head -> next; // aranan veri yoksa liste taranıyor
        }
        else {
            locate = head;
            break; // veri bulunursa döngüden çıkılarak geri döndürülüyor
        }
    }
    return locate;
}
```

Çift Ba lı Do rusal Listede Kar ıla tırma Yapmak

Verilen node'un bu listede var olup olmadığını kontrol eden fonksiyondur. Fonksiyon e er listede node varsa 1, yoksa 0 ile geri döner.

```
bool is_member(struct node* other_node, struct node* head) {
    while(head != NULL && head != other_node)
        head = head -> next;
    return(head == other_node); // ifade do ruysa 1, de ilse 0 geri döndürülür.
}
```

Verilen örneklerdeki fonksiyonlar, varsayılan bir listenin yahut global tanımlanmış head de i keninin varlığı, yine global olarak tanımlanmış yapıların olduğu kabul edilerek yazılmıştır. E er bu kabulümüz olmasaydı üphesiz kontrol ifadelerini de içeren uzun kod satırları meydana gelirdi.

Çift Ba lı Do rusal Listelerin Avantajları ve Dezavantajları

Avantajları

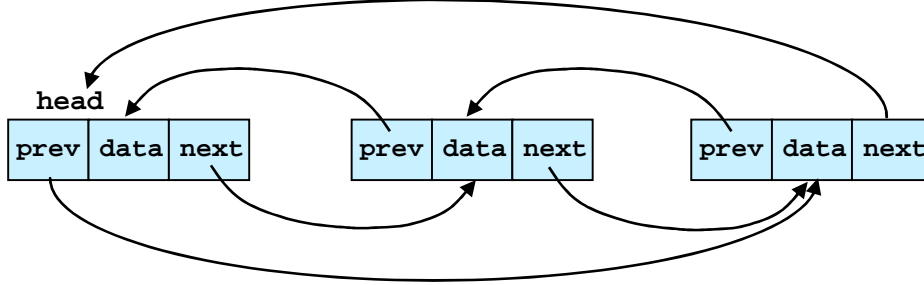
- Her iki yönde gezilebilir,
- Ekleme,Silme gibi bazı i lemler daha kolaydır.

Dezavantajları

- Bellekte daha fazla yer kaplar,
- Her dü ümün prev ve next adında iki i aretçisi olduğu için liste i lemleri daha yava tır,
- Hata olasılığı yüksektir. Örne in dü ümlerin prev i aretçisinin bir önceki dü üme ba lanması ya da next i aretçisinin bir sonraki dü üme ba lanması unutulabilir.

2.9 Çift Ba lı Dairesel (Double Linked) Listeler

iki ba lı dairesel listelerde listenin birinci dü ününün prev de eri ve sonuncu dü ününün next de eri NULL olmaktadır. Ancak iki ba lı dairesel listelerde listenin birinci dü ününün prev de eri sonuncu dü ününün data kısmını, sonuncu dü ününün next de eri de listenin ba mını (yani listenin ismini) göstermektedir. Fonksiyonlarda bununla beraber de i mektedir.



ekil 2.10 Çift ba lı dairesel listeler.

Çift Ba lı Dairesel Listelerde Ba a Dü üm Ekleme

Listenin ba mına bir dü üm ekleyen insertAtFirst fonksiyonunu yazalım.

```
void addhead(struct node*&head, int key) {
    if(head == NULL) {
        head = (struct node *)malloc(sizeof(struct node));
        head -> data = key;
        head -> next = head;
        head -> prev = head;
    }
    else {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> data = key;
        struct node *last = head;
        // liste çift ba lı ve dairesel oldu u için son eleman head->prev dir.
        head->prev->next=temp;
        temp->next=head;
        temp->prev=head->prev;
        head->prev=temp;
        head = temp;
    }
}
```

Çift Ba lı Dairesel Listenin Sonuna Eleman Ekleme

addhead fonksiyonundaki son satır head = temp; yazılmaz ise listenin sonuna ekleme yapılmı olur.

```
void addlast(struct node* temp, struct node*&head) {
    if(!head)
        head = temp;
    else {
        temp -> next = last(head) -> next;
        temp -> prev = last(head);
        last(head) -> next = temp; // last fonksiyonu ile son dü üm bulunuyor
        head -> prev = temp;
    }
}
```

Çift Ba lı Dairesel Listelerde ki Listeyi Birle tirmek

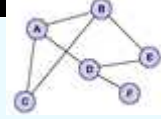
```
// list_1 listesinin sonuna list_2 listesini eklemek
void concatenate(struct node*& list_1, struct node* list_2){ //parametrelere dikkat
    if(list_1 == NULL)
        list_1 = list_2;
    else {
        // Birinci listenin son dü ümünü last olarak bulmak için
        struct node *last=list_1;
        while(last -> next != list_1)
            last = last -> next;
        last->next=list_2; //Birinci listenin sonu ikinci listenin başına ba landı
        list2->prev=last; //İkinci listenin başı birinci listenin sonuna ba landı
        // İkinci listenin son dü ümünü last olarak bulmak için
        last=list_2;
        while(last -> next != list_2)
            last = last -> next;
        last->next=list_1; //İkinci listenin sonu birinci listenin başına ba landı
        list1->prev=last; //Birinci listenin başı ikinci listenin sonuna ba landı
    }
}
```

Çift Ba lı Dairesel Listelerde Araya Eleman Eklemek

```
// head listesinin n. dü ümünün hemen ardına other_node dü ümünü ekleme
void addthen(struct node* other_node, struct node*&head, int n) {
    node* temp = head;
    int i = 1;

    while(i < n) {
        head = head -> next;
        i++;
    }

    head -> next -> prev = other_node;
    other_node -> prev = head;
    other_node -> next = head -> next;
    head -> next = other_node;
    head = temp;
}
```

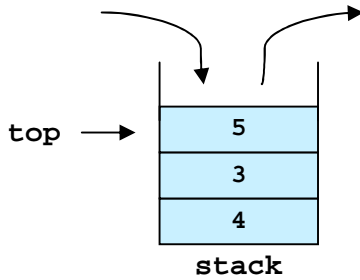


BÖLÜM

Yığınlar (Stacks) 3

3.1 YIĞINLARA (Stacks) GİRİŞ

Veri yapılarının önemli konularından birisidir. Nesne ekleme ve çıkarmalarının en üstten (*top*) yapıldığı bir veri yapısına **stack** (*yığın*) adı verilir. Soyut bir veri tipidir. Bazı kaynaklarda *yığın*, nadiren *çıkım* olarak da isimlendirilir. Bir nesne ekleneceği zaman yığın *ın* en üstüne konulur. Bir nesne çıkarılacağı zaman da yığın *ın* en üstündeki nesne çıkarılır. Bu çıkarılan nesne, yığın *ın* elemanlarının içindeki en son eklenen nesnedir. Yani bir yığın *ındaki* elemanlardan sadece en son eklenene erişim yapılır. Bu nedenle yığın *lara* **LIFO** (*Last In First Out: Son giren ilk çıkar*) listesi de denilir. Mantıksal yapısı bir konteyner gibi düşünülebilir.



ekil 3.1 Bir stack'ın mantıksal yapısı.

ekilde görüldüğü gibi, nesnelere *son giren ilk çıkar* (**LIFO**) prensibine göre eklenip çıkarılıyor. *top* isimli değişken ise en üstteki nesnenin indisini tutuyor.

Stack yapılarına gerçek hayattan benzetmeler yapacak olursak;

- arjör,
- yemekhanedeki tepsi,
- el feneri pilleri,

gibi son girenin ilk çıktığı örnekler verilebilir. Bilgisayar'da nerelerde kullanıldığını ilerleyen sayfalarda göreceğiz.

Yığınlar statik veriler olabileceği gibi, dinamik veriler de olabilirler. Bir yığın statik olarak tanımlanırken dizi eklinde, dinamik olarak tanımlanırken ise basılı liste biçiminde tanımlanabilir. İmdi stack'lerin C programlama dilini kullanarak bilgisayarlardaki implementasyonunun nasıl gerçekleştirildiğini görelim.

3.2 STACK'LERİN DİZİ (Array) İMPLEMENTASYONU

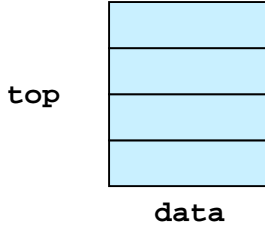
Stack'lerin bir boyutu (*kapasitesi*) olacağından önce bu boyutu tanımlamalıyız.

```
#define STACK_SIZE 4
```

C derleyicilerinin `#define` önilemcisi komutunu biliyor olmalıyız. C programlama dilinde `#` ile başlayan bütün satırlar, önilemci programa verilen komutlardır (*directives*). Üstteki satır bir anlamda derleyiciye `STACK_SIZE` gördüğün yere 4 yaz demektir. Program içerisinde dizi boyutunun belirtilmesi gereken alana bir sabit ifadesi olan 4 rakamını de il `STACK_SIZE` yazacağız. Böyle bir tanımlama ile uzun kodlardan oluşabilecek program içerisinde, dizi boyutunun değiştirilmesi ihtiyacı olduğunda 4 rakamını güncellemek yeterli olacaktır ve programın uzun kod satırları arasında dizi boyutunu tek tek değiştirmek zahmetinden kurtaracaktır. İmdi bir stack'ın genel yapısını tanımlayalım;

```
#define STACK_SIZE 4
typedef struct {
    int data[STACK_SIZE]; // ihtiyaca göre veri türü de işebilir
    int top;
    /* süreklı eleman ekleme ve silme işlemi yapılacak için en üstteki
     * elemanın indisini tutan top adında bir de işken tanımladık */
}stack;
```

Bu tanımlama ile meydana gelen durum a a ıdaki ekilde görülmektedir.



ekil 3.2 LIFO ihtiyacı için kullanılacak veri yapısı.

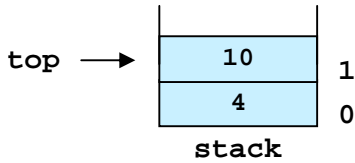
Dikkat ederseniz `top` de i ken i u an için herhangi bir `data` 'nın indisini tutmuyor. C derleyicisi tarafından bir çöp de er atanmı durumdadır. `stack`'e ekleme i lemi yapan `push` isimli fonksiyonumuzu yazalım.

Stack'lere Eleman Ekleme i lemi (`push`)

Bir `stack`'e eleman ekleme fonksiyonu `push` olarak bilinir ve bu fonksiyon, `stack`'i ve eklenecek elemanın de erini parametre olarak alır. Geri dönü de eri olmayaca ı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için y ı mın dolu olmaması gerekir.

```
void push(stack *stk, int c) {
    if(stk -> top == STACK_SIZE - 1) // top son indisi tutuyorsa doludur
        printf("\nStack dolu\n\n");
    else {
        stk -> top ++;
        stk -> data[stk -> top] = c;
    }
}
```

Eleman eklendikten sonra `top` de i ken i en son eklenen elemanın indis de erini tutmaktadır.

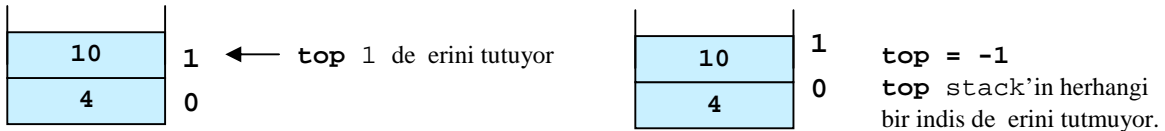


ekil 3.3 `top` de i ken i 1 indis de erini tutuyor.

Bir Stack'in Tüm Elemanlarını Silme i lemi (`reset`)

`stack`'i resetlemek için `top` de i keninin de erini `-1` yapmamız yeterlidir. Bu atamadan sonra `top` herhangi bir indis de eri tutmuyor demektir. Aslında veriler silinmedi de il mi? Bu veriler hala hafızada bir yerlerde tutuluyor. Lakin `stack`'e hiç ekleme yapılmamı bile olsa `stack`'lerin dizi ile implementasyonu bellekte yine de yer i gal edecekti. Yeni eklenecek veriler ise öncekilerinin üzerine yazılacaktır. imdi `reset` isimindeki fonksiyonumuzu yazalım.

```
void reset(stack *stk) {
    stk -> top = -1;
}
```



ekil 3.4 Stack'lerde resetleme i lemi.

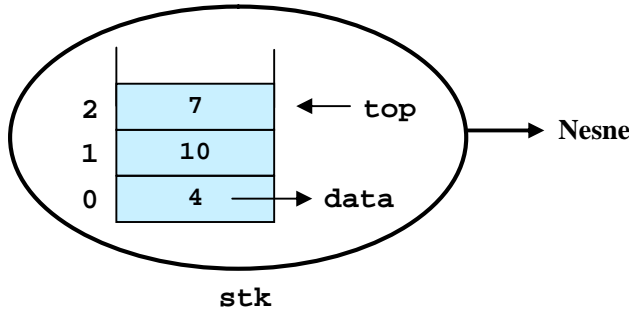
Stack'lerden Eleman Çıkarma i lemi (`pop`)

`pop` fonksiyonu olarak bilinir. Son giren elemanı çıkarmak demektir. Çıkarılan elemanın indis de eriyle geri dönece i için fonksiyonun tipi `int` olmalıdır.

```
int pop(stack *stk) {
    if(stk -> top == -1) // stack boş mu diye kontrol ediliyor
        printf("Stack bos");
    else {
        int x = stk -> top--; // -- operatörünün işlem sırasına dikkat
        return x; // çıkarılan elemanın indis de eriyle geri dönüyor
    }
}
```

else kısmı u tek satır kodla da yazılabilirdi. Fakat bu kez geri dönüşünün stack'ten çıkarılan verinin kendisi oldu una dikkat ediniz.

```
else
    return(stk -> data[stk -> top--]);
```



ekil 3.5 stk nesne modeli.

Aşağıda `top()` ve `pop()` fonksiyonlarının `main()` içerisindeki kullanımını görüyorsunuz. stack yapısının ve fonksiyon tanımlamalarının `main()` bloğunda yapıldığını varsayıyoruz.

```
int main() {
    int x;
    stack n;
    reset(&n);

    push(&n, 4);
    push(&n, 2);
    push(&n, 5);
    push(&n, 7);
    push(&n, 11);

    x = pop(&n);
    printf("%d\n", x);

    x = pop(&n);
    printf("%d\n", x);

    x = pop(&n);
    printf("%d\n", x);

    x = pop(&n);
    printf("%d\n", x);

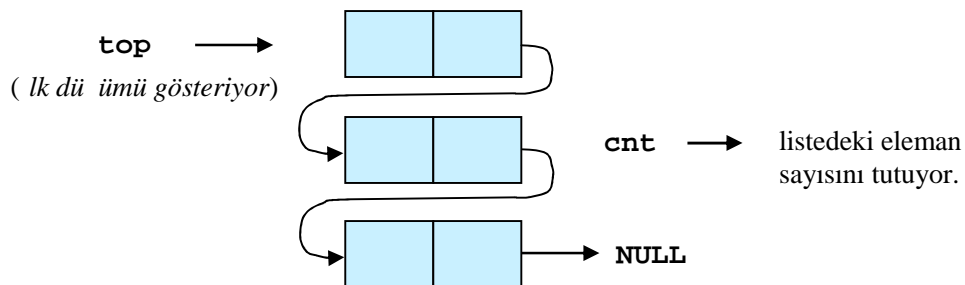
    x = pop(&n);
    printf("%d\n", x);

    getch();
    return 0;
}
```

Alıştırma-2: Yukarıdaki kod satırlarında her `pop` işleminden sonra `return` edilen `x` değerinin son `pop` işleminden sonraki değeri ne olur?

3.3 STACK'LERİN BAĞLI LİSTE (Linked List) İMPLEMENTASYONU

Stack'in bağlı liste uygulamasında, elemanlar bir yapının içinde yapıları ile beraber bulunur. Mantıksal yapısını daha önce gördüğümüz bağlı listelerin mantıksal yapısının üst üste sıralanmış ekli gibi düşünebilirsiniz.



ekil 3.6 Bağlı liste kullanılarak gerçekleştirilen bir stack'in mantıksal yapısı.

Dizi uygulamasının verdi i maksimum genilik sınırı kontrolü bu uygulamada yoktur. Tüm fonksiyonlar sabit zamanda gerçekleşir. Çünkü fonksiyonlarda `stack`'in genili referans alınır. Hemen hemen bütün fonksiyonların görevleri, dizi uygulamasındaki görevleriyle aynıdır. Fakat iki uygulama arasında algoritma farkı olduğu için fonksiyon tanımlamaları da farklı olur. Tek başlı olarak dizi listesi kullanarak veri yapısını yazalım.

```
typedef struct {
    struct node *top;
    int cnt;
}stack;
```

Görüldüğü üzere `stack`, liste uzunluğunu tutacak `int` türden bir `cnt` değeri ve `struct node` türden göstericisi olan bir yapıdır. `stack`'in tanımı `typedef` bildirimleriyle yapılmıştır. Bu yapıdan nesnelere oluşturulması zaman zaman `struct stack` tür belirtimi yerine yalnızca `stack` yazılması geçerli olacaktır.

Stack'in Bo Olup Olmadığının Kontrolü (`isEmpty`)

C'de enumeration (*numaralandırma*) konusunu hatırlıyor olmalısınız. Biz de ilk olarak enumeration yoluyla boolean türünü tanımlıyoruz. C++'ta ise boolean türü öntanımlıdır. Ayrıca tanımlamaya gerek yoktur.

```
typedef enum {false, true}boolean;
```

`typedef` bildiriminden sonra `boolean`, yalnızca `false` ve `true` değerlerini alabilen bir türün ismidir. `false` 0 değerini, `true` ise 1 değerini alacaktır. Artık `enum` anahtar sözcüğü kullanılmadan direkt `boolean` türden tanımlamalar yapılabilir. Aşağıda `isEmpty` isimli fonksiyonu görüyorsunuz.

```
boolean isEmpty(stack *stk) {
    return(stk -> cnt == 0); // parantez içerisindeki ifadeye dikkat
}
```

Fonksiyonun geri dönüş değeri türü `boolean` türüdür. Geri döndürülen değer sadece yanlış veya doğru olabilir. `return` parantezi içerisindeki ifade ile `stk -> cnt` değeri 0'a eşitse `true`, eşit değilse `false` değeri geri döndürülecektir.

Stack'in Dolu Olup Olmadığının Kontrolü (`isFull`)

Yine geri dönüş değeri türü `boolean` türden olan `isFull` isimli fonksiyonu tanımlıyoruz. `stack` yapısının `cnt` değeri keninin değeri, `stack`'in boyutunu sınırlandıran `STACK_SIZE` değeri kenine eşitse dolu demektir ve fonksiyondan geriye `true` döndürülür. Eşit değilse geri dönüş değeri `false` olacaktır.

```
boolean isFull(stack *stk) {
    return(stk -> cnt == STACK_SIZE);
}
```

Stack'lere Yeni Bir Düzüm Ekleme (`push`)

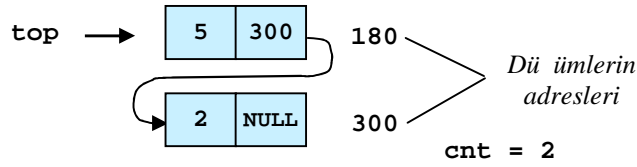
Tek başlı liste uygulamasında da `stack`'lerde ekleme işlemi yapan fonksiyon `push` fonksiyonu olarak bilinir. Dizi uygulamasındaki `push` fonksiyonuyla bazı farkları vardır. `stack`'lerin dizi implementasyonunda dizi boyutu önceden belirlenir. Dizi boyutu aşıldığında ise taşma hatası meydana gelir. Oysa tek başlı liste uygulaması ile oluşturulan `stack`'lerde taşma hatası meydana gelmez. Aslında bellekte boş yer olduğu sürece ekleme yapılabilir. Fakat `stack`'in soyut veri yapısından dolayı sanki bir büyüklüğü varmış gibi ekleme yapılır. Yeni bir düzüm ekleneceğinden dolayı fonksiyon içerisinde `struct node` türden bir yapı oluşturulması gerekecektir.

```
// stack'e ekleme yapan fonksiyon
void push(stack *stk, int c) {
    if(!isfull(stk)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        /* struct node *temp = new node(); // C++'ta bu şekilde */

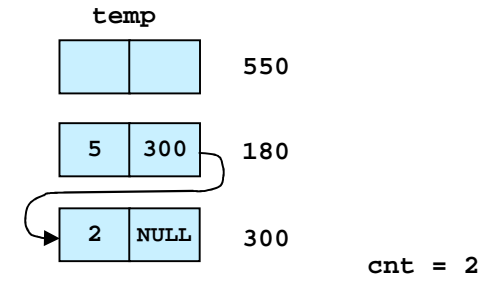
        temp -> data = c;
        temp -> next = stk -> top;
        stk -> top = temp;
        stk -> cnt++;
    }
    else
        printf("Stack dolu\n");
}
```

Görüldüğü gibi daha önce anlatılan tek başlı listelerde eleman ekleme işleminden tek farkı `stk -> cnt++` ifadesidir. Yapılan işlemler aşağıdaki şekilde adım adım belirtilmiştir.

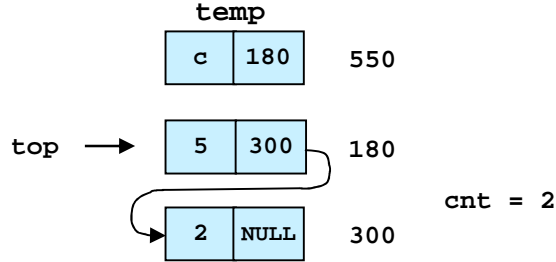
top u anda 180
adresini gösteriyor



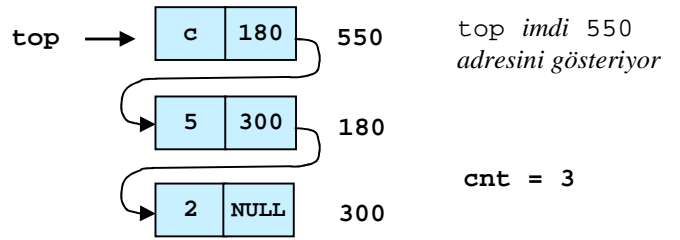
Adım 1: stack'in başlangıçtaki durumu.



Adım 2: temp dü üümü için bellekte yer ayrıldı.



Adım 3: Yeni dü üme veriler atandı.



Adım 4: top artık listenin başını gösteriyor.

ekil 3.7 Başlı liste kullanılarak oluşturulmuş bir stack'e yeni bir dü üm eklemek.

Stack'lerden Bir Dü üümü Silmek (pop)

Yine dizi uygulamasında oldu u gibi pop fonksiyonu olarak bilinir. Silinecek dü ümü belle e geri kazandırmak için struct node türden temp adında bir i aretçi tanımlanarak silinecek dü ümün adresi bu i aretçiye atanır. top göstericisine ise bir sonraki dü ümün adresi atanır ve temp dü üümü free() fonksiyonuyla silinerek belle e geri kazandırılır. Eleman sayısını tutan cnt de i keninin de eri de 1 azaltılır. Fonksiyon silinen dü ümün verisiyle geri dönce i için türü int olmalıdır.

```
int pop(stack *stk) {
    if(!isEmpty(stk)) { // stack boş de ilse
        struct node *temp = stk -> top;
        stk -> top = stk -> top -> next;
        int x = temp -> data;
        free(temp);
        stk -> cnt--;
        return x;
    }
}
```

Fonksiyon, e er stack boş de ilse dü üm silme i lemini gerçekleştiriyor ve silinen dü ümün verisini geri döndürüyor. stack'in boş olması halinde fonksiyonun else bölümüne ekrana uyarı mesajı veren bir kod satırı da eklenebilirdi.

Stack'in En Üstteki Verisini Bulmak (top)

stack boş de ilse en üstteki dü ümün verisini geri döndüren bir fonksiyondur. Geri dönu de eri türü, verinin türüyle aynı olmalıdır.

```
int top(stack *stk) {
    if(!isEmpty(stk))
        return(stk -> top -> data); // En üstteki elemanın verisiyle geri döner
}
```

Bir Stack'e Başlangıç De erlerini Vermek (initialize)

Önemli bir fonksiyondur. De i kenlere bir de er ataması yapılmadı ı zaman ço u C derleyicisi bu de i kenlere, çöp de er diye de tabir edilen rastgele de erler atayacaktır. Yı ın i lemlerinden önce mutlaka bu fonksiyonla başlangıç de erleri verilmelidir.

```
void initialize(stack *stk) {
    stk -> top = NULL;
    stk -> cnt = 0;
}
```

Stack'ler Bilgisayar Dünyasında Nerelerde Kullanılır?

Yığın mantığı bilgisayar donanım ve yazılım uygulamalarında yaygın olarak kullanılmaktadır. Bunlardan bazıları;

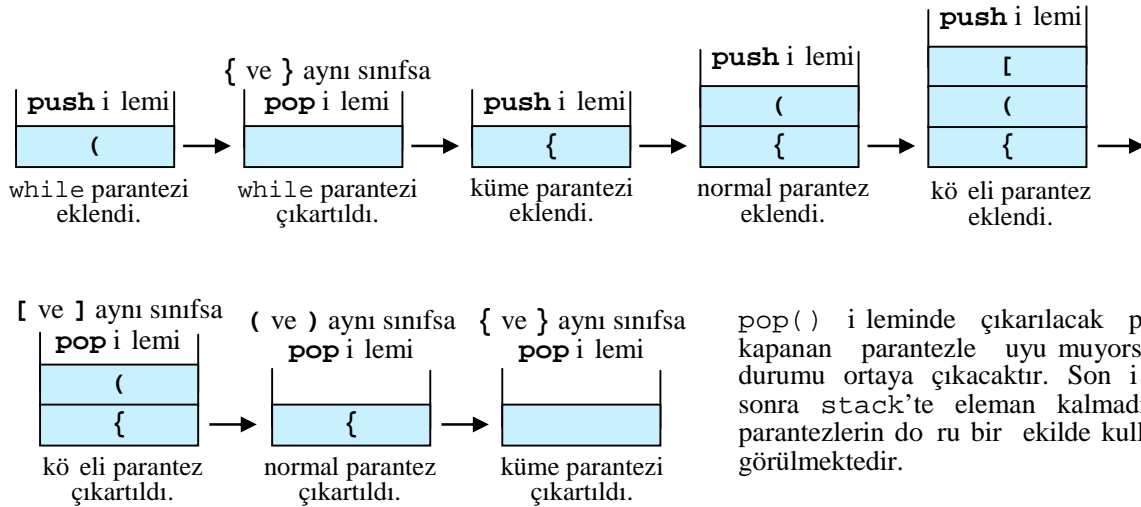
- Rekürsif olarak tanımlanan bir fonksiyon çalışırken hafıza kullanımı bu yöntem ile ele alınır,
- (, { , [,] , } ,) ayrıaçlarının C/C++ derleyicisinin kontrollerinde,
- postfix \rightarrow infix dönüşümlerinde,
- Yazılım uygulamalarındaki Parsing ve Undo işlemlerinde,
- Web browser'lardaki Back butonu (önceki sayfaya) uygulamasında,
- Ayrıca, mikroilemcinin içyapısında **stack** adı verilen özel hafıza alanı ile mikroilemci arasında, bazı program komutları ile (*push* ve *pop* gibi), bir takım işlemlerde (*alt program çağırma* ve *kesmeler* gibi), veri transferi gerçekleşir. Mikroilemcinin içinde, hafızadaki bu özel alanı gösteren bir yığın işaretçisi (*Stack Pointer –SP*) bulunur. Stack Pointer o anda bellekte çalışılan bölgenin adresini tutar. *push* fonksiyonu, stack'e veri göndermede (*yazmada*), *pop* fonksiyonu ise bu bölgeden veri almada (*okumada*) kullanılır.

Bölüm 1'deki Özyinelemeli Fonksiyonlar konusunda gösterilen rekürsif faktöryel uygulamasında, fonksiyonun kendisini her çağırmasında stack'e ekleme yapılır. stack'in en üstünde *fact(0)* bulunmaktadır. Fonksiyon çağırıldıktan sonra her geri döndüğünde ise stack'ten çıkarma işlemi yapılmaktadır.

Örnek 3.1: Anlamsız bir kod parçası olsun;

```
while(x == 7) {
    printf("%d", a[2]);
}
```

Üstteki kodlarda bulunan parantezlerin kullanımlarının doğruluğu stack'lerle kontrol edilir. Bunun nasıl yapıldığını alttaki ekilde adımlar halinde görülmüyor. Bu uygulamada yukarıdaki kod parçasında bulunan bir parantez açıldığında *push()* fonksiyonuyla stack'e ekleme, kapatıldığında ise *pop()* fonksiyonuyla çıkarma yapacağız.



ekil 3.8 Parantez kontrolünün ekilsel gösterimi.

3.4 INFIX, PREFIX VE POSTFIX NOTASYONLARI

Bilgisayarlarda infix yazım türünün çözülmesi zordur. Acaba $x = a/b - c + d * e - a * c$ eklindeki bir ifadeyi çözümlerken, $((4/2) - 2) + (3 * 3) - (4 * 2)$ gibi bir ifadenin degerini hesaplarırken ya da $a / (b - c) + d * (e - a) * c$ gibi parantezli bir ifadeyi değerlendirirken derleyiciler sorunun üstesinden nasıl geliyor? $3 * (55 - 32 - (11 - 4) + (533 - (533 - (533 + (533 - (533 + 212)))) * 21 - 2))$ gibi birçok operatör ve operand içeren bir işlemde nasıl operatör önceliklerine göre işlem sıralarını doğru belirleyip sonuç üretebiliyorlar?

Bir ifadeye farklı önceliklere sahip operatörler yazılma sırasıyla işlemirse ifade yanlış sonuçlandırılabilir. Örneğin $3 + 4 * 2$ ifadesi $7 * 2 = 14$ ile sonuçlandırılabilirken $3 + 8 = 11$ ile de sonuçlandırılabilir. Bilgisayarlarda infix yazım türünün çözülmesi zordur. Bu yüzden ifadelerin operatör önceliklerine göre ayrıştırılması, ayrılan parçaların sıralanması ve bu sıralamaya uyularak işlem yapılması gerekir. Bu işlemler için prefix ya da postfix notasyonu kullanılır. Çoğu derleyici, kaynak kod içerisinde infix notasyonunu kullanmaktadır ve daha sonra stack veri yapısını kullanarak prefix veya postfix notasyonuna çevirir.

Bu bölümde bilgisayar alanındaki önemli konulardan biri olan infix, prefix ve postfix kavramları üzerinde duracağız ve bu kavramlarda stack kullanımını göreceğiz.

- Operatörler : +, -, /, *, ^
- Operandlar : A, B, C... gibi isimler ya da sayılar.

Infix notasyonu: Alı a geldi imiz ifadeler infix eklindedir. Operatörlerin i lenecek operandlar arasına yerle tirildi i gösterim biçimidir. Bu gösterimde operatör önceliklerinin de i tirilebilmesi için parantez kullanılması arttır. Örne in infix notasyonundaki $2+4*6$ ifadesi $2+24=26$ ile sonuçlanır. Aynı ifadede + operatörüne öncelik verilmesi istenirse parantezler kullanılır; $(2+4)*6$. Böylece ifade 36 ile sonuçlandırılır.

Prefix notasyonu: Prefix notasyonunda (PN, *polish notation*) operatörler, operandlarından önce yazılır. Örne in $2+4*6$ ifadesi infix notasyonundadır ve prefix notasyonunda $+2*46$ ekinde gösterilir. Benzer biçimde $(2+4)*6$ ifadesi $*+246$ ekinde gösterilir. Görüldü ü gibi prefix notasyonunda i lem önceliklerinin sa lanması için parantezlere ihtiyaç duyulmamaktadır.

Postfix notasyonu: Postfix notasyonunda (RPN, *reverse polish notation*) ise önce operandlar ve ardından operatör yerle tirilir. Aynı örnek üzerinden devam edersek; infix notasyonundaki $2+4*6$ ifadesi prefix notasyonunda $2 4 6 * +$ ekinde, benzer biçimde $(2+4)*6$ ifadesi de $2 4 + 6 *$ ekinde gösterilir. Yine prefix'te oldu u gibi bu gösterimde de parantezlere ihtiyaç duyulmamaktadır. Bazı bilgisayarlar matematiksel ifadeleri postfix olarak daha iyi saklayabilmektedir.

Tüm aritmetik ifadeler bu gösterimlerden birini kullanarak yazılabilir. Ancak, bir yazmaç (*register*) yı mı ile birle tirilmi postfix gösterimi, aritmetik ifadelerin hesaplanmasında en verimli yoldur. Aslında bazı elektronik hesap makineleri (*Hewlett-Packard gibi*) kullanıcının ifadeleri postfix gösteriminde girmesini ister. Bu hesap makinelerinde biraz alı tırma yapıldı nda, iç içe birçok parantez içeren uzun ifadeleri, terimlerin nasıl gruplandı mı bile dü ünmeden, hızlı bir ekinde hesaplamak mümkündür.

- lem önceli i;
- 1- Parantez içi
- 2- Üs alma
- 3- Çarpma/Bölme
- 4- Toplama Çıkarma

Aynı önceli e sahip i lemlerde sıra soldan sa a () do rudur. Yalnız üs almada sa dan sola do ru i lem yapılır.

Infix	Prefix	Postfix
$A+B-C$	$-+ABC$	$AB+C-$
$(A+B)*(C-D)$	$*+AB-CD$	$AB+CD-*$
$A/B^C+D^*E-A^*C$	$-+/A^BC*DE*AC$	$ABC^/DE^+AC*-$
$(B^2-4^*A^*C)^{(1/2)}$	$(^-^B2**4AC/12)$	$(B2^4A^*C*-12/^)$
$A^B^*C-D+E/F/(G+H)$	$+-*ABCD//EF+GH$	$AB^C^*D-EF/GH+/+$
$((A+B)^*C-(D-E))^{(F+G)}$	$^--*+ABC-DE+FG$	$AB+C^*DE-FG+^$
$A-B/(C^*D^E)$	$-A/B^*C^DE$	$ABCDE^*/-$

Tablo 3.1 Matematiksel ifadelerde infix, prefix ve postfix notasyonunun gösterimi.

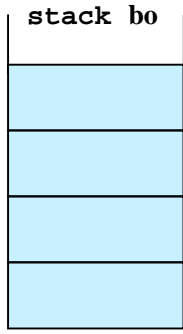
Dikkat edilecek olunursa, postfix ile prefix ifadeler birbirinin ayna görüntüsü de illerdir. imdi bir örnekle notasyon i lemlerinin stack kullanılarak nasıl gerçekte tirildiklerini görelim.

Örnek 3,2: $3 2 * 5 6 * +$ postfix ifadesini stack kullanarak hesaplayınız.

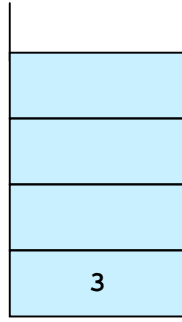
Çözüm algoritması öyle olmalıdır;

- 1- Bir operandla kar ıla ıldı nda o operand stack'e eklenir,
- 2- Bir operatör ile kar ıla ıldı nda ise o operatörün gerek duydu u sayıda operand stack'ten çıkarılır,
- 3- Daha sonra pop edilen iki operanda operatör uygulanır,
- 4- Bulunan sonuç tekrar stack'e eklenir.

Bu adımları uygulayarak örne in çözümünü ekillerle açıklayalım.

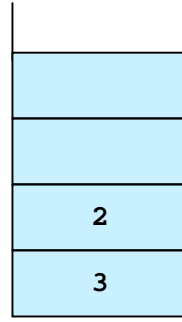


push
→



3 sayısı stack'e eklendi.

push
→

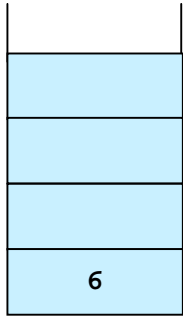


2 sayısı stack'e eklendi.

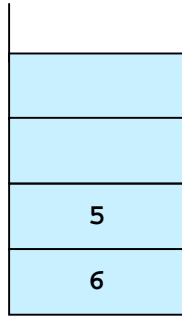
pop
→

imdi bir operatörle karılaıldı ı için stack'teki elemanlara 2 defa pop i lemi uygulandıktan sonra operatörle i leme sokulup sonuç push yapılacak.

Bo bir stack. Bir operatörle karıla ıncaya kadar push yapılacak.

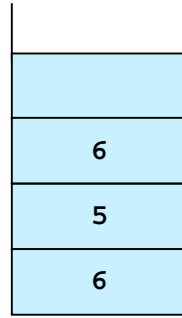


push
→



5 sayısı stack'e eklendi.

push
→

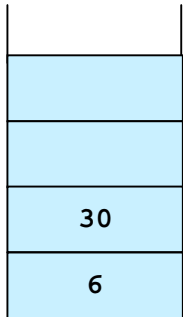


6 sayısı stack'e eklendi.

pop
→

Çarpı operatörüyle karılaıldı ı için stack'teki elemanlara 2 defa pop i lemi uygulandıktan sonra operatörle i leme sokulup sonuç push yapılacak.

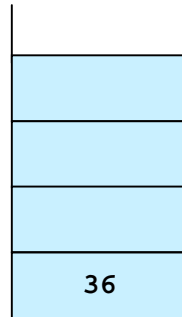
Çarpım sonucu olan 6 sayısı stack'e eklendi.



pop
→

Artı operatörüyle karılaıldı ı için stack'teki elemanlara 2 defa pop i lemi uygulandıktan sonra operatörle i leme sokulup sonuç push yapılacak.

push
→



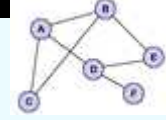
36 sayısı stack'e eklendi.

pop
→

Artık hesaplanacak bir i lem kalmadı ı için sonuç ekrana yazdırılabilir.

Çarpım sonucu olan 30 sayısı stack'e eklendi.

ekil 3.9 Postfix ifadesinin stack kullanılarak hesaplanması.



B Ö L Ü M

Queues (Kuyruklar)

4

4.1 G R

Kuyruk (*queue*) veri yapısı *stack* veri yapısına çok benzer. Kuyruk veri yapısında da veri ekleme (*enqueue*) ve kuyruktan veri çıkarma (*dequeue*) ekinde iki tane i lem tanımlıdır. Yı nların tersine FIFO (*First In First Out*) prensibine göre çalı ırlar ve ara elemanlara eri im do rudan yapılamaz. Veri ekleme *stack*'teki gibi listenin sonuna eklenir. Fakat veri çıkarılaca ı zaman listenin son elemanı de il ilk elemanı çıkarılır. Bu ise, kuyruk veri yapısının ilk giren ilk çıkar tarzı bir veri yapısı oldu u anlamına gelir.

Kuyru un çalı ma mantı nı anlatan gündelik hayatta kar ıla tı ımız olaylara birkaç örnek verirsek;

- sinema gi esinden bilet almak için bekleyen insanlar,
- bankamatikten para çekmek için bekleyen insanlar,

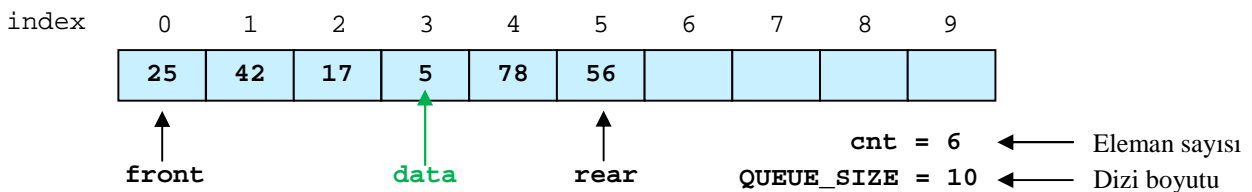
gibi, kuyru a ilk gelen ki i ilk hizmeti alır (*ilk i lem görür*) ve kuyruktan çıkar.

Kuyruk (*queue*) yapısı bilgisayar alanında; a , i letim sistemleri, istatistiksel hesaplamalar, simülasyon ve çoklu ortam uygulamalarında yaygın olarak kullanılmaktadır. Örne in, yazıcı kuyruk yapısıyla i lemleri gerçekle tirmektedir. Yazdır komutu ile verilen birden fazla belgeden ilki yazdırılmakta, di erleri ise kuyrukta bekletilmekte, sırası geldi inde ise yazdırılmaktadır.

Kuyruk veri yapısı bir sabit dizi olarak gösterilebilir veya bir ba lı liste olarak tanımlanabilir. Kuyrukların array implementasyonlarında, kuyru un eleman sayısını tutan bir de i ken ile kuyru un ba nı gösteren (*genellikle front olarak isimlendirilir*) ve kuyru un sonunu gösteren (*genellikle rear olarak isimlendirilir*) üç adet *int* türden de i ken ile saklanacak verinin türüne uygun bir dizi bulunur. Kuyrukların ba lı liste implementasyonlarında ise kuyru un ba nı ve sonunu gösteren *struct node* türden iki i aretçi ile eleman sayısını tutan *int* türden bir counter'ı bulunur. Kuyru un ba nı gösteren i aretçi her veri çıkarıldı nda bir sonraki veriyi gösterir. Kuyru un sonunu gösteren i aretçi ise her veri eklendi inde yeni gelen veriyi gösterecek ekinde de i tirilir.

Kuyruk (*queue*) yapısında da yı n yapısında oldu u gibi eleman ekleme ve eleman çıkarma olmak üzere iki i lem söz konusudur. Kuyru a eleman eklemek için kuyru un dolu olmaması gerekir ve ilk olarak kuyru un dolu olup olmadı ı kontrol edilir ve bo yer varsa eleman eklenir; bo yer yoksa ekleme i lemi ba arısız olur. Kuyru a eleman ekleme i lemi **ENQUEUE** olarak adlandırılır. Kuyruktan eleman çıkarmak için de kuyru un bo olmaması gerekir. Bu çıkarma i lemi için de ilk olarak kuyru un bo olup olmadı ı kontrol edilir ve kuyruk bo de ilse, eleman çıkarma i lemi gerçekle tirilir. E er bo sa eleman çıkarma i lemi ba arısız olur. Kuyruktan eleman çıkarma i lemi **DEQUEUE** olarak bilinir.

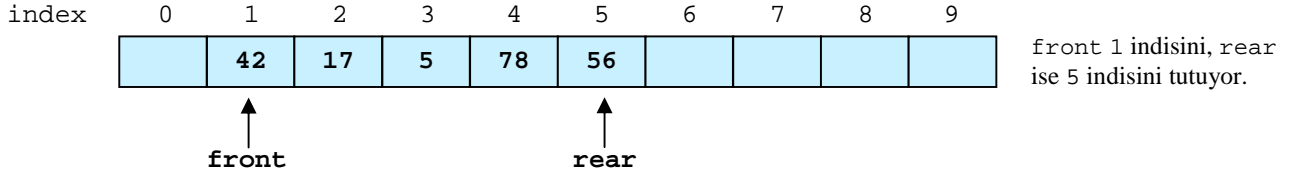
4.2 KUYRUKLARIN D Z (Array) MPLEMENTASYONU



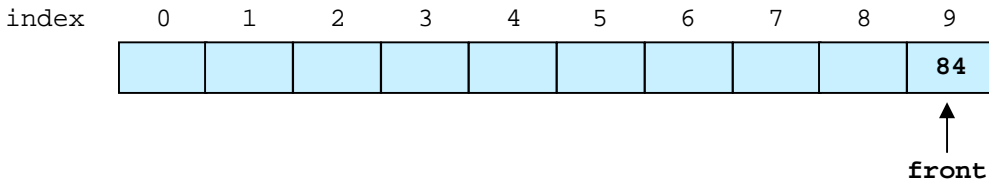
ekil 4.1 Kuyruklarda (*queue*) array implementasyonunun mantıksal gösterimi.

Mü teri kuyruklarında öndeki mü teri bekledi i hizmeti alıp kuyruktaki yerinden ayrılınca, kuyruktakiler birer adım kolayca, ön tarafa istekli biçimde yürürler. Yazılım içinde olu turulmu olan kuyruklarda ise bu i lem kendili inden ve kolay uygulanır bir nitelik de ildir. Basit yapılı kuyruk denen bu modelin sakıncası, kuyruktaki ö elerin sayısı kadar kaydırma i leminin kuyruktan her ayrılma olayında yapılmasıdır. Ekleme ve çıkarma i lemlerinde ön indeks (*front*) daima sabittir. Çıkarma (*dequeue*) i leminde, kuyru un önündeki eleman çıkarılır ve di er bütün elemanlar bir öne kayar, ayrıca *rear* de i keninin de eri de 1 azaltılır. Ekleme (*enqueue*) i leminde ise arka indeks (*rear*) dizide ileri do ru hareket eder. ekil 4.1'de basit yapılı bir kuyruk modeli verilmi tir.

Sürekli kaydırma i leminin maliyeti dü ünüldü ün de eleman çıkarmalarda verileri kaydırmak yerine **front** ön indeksi kaydırılсын eklinde bir çözüm akla gelebilir. Fakat bu defa da ba ka bir problem ortaya çıkmaktadır. Yukarıdaki ekil dikkate alındı nda kuyruklarda enqueue i lemi sa taraftan, dequeue i lemi ise sol taraftan yapılmaktadır. ekil 4.2’de görüldü ü gibi verileri kaydırmayıp **front** indeksini kaydıracağız a göre kuyru un solundan bir eleman çıkartıldı nda orada boşluk oluşacak, **front** ise 1 no’lu indisi gösterecektir. Sürekli bir dequeue i leminde ba ka bir önlem alınmazsa **front** indeksi dizi sonuna kadar ulaşabilecektir. Bu durumu da ekil 4.3’te görüyorsunuz.

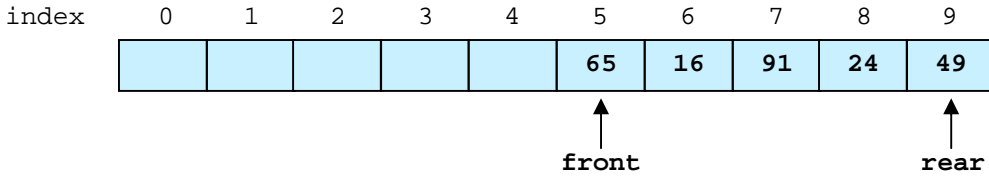


ekil 4.2 dequeue i lemi sonrası kuyru un görünümü.



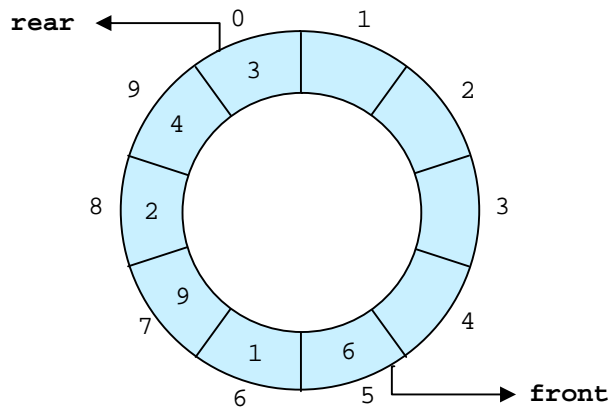
ekil 4.3 front indeksi kuyru un son indisini gösteriyor.

imdi ekil 4.2’deki duruma göre birkaç defa enqueue ve dequeue i lemleri sonucunda ekil 4.4’te görüldü ü gibi rear indeksinin kuyru un son indisine kadar geldi ini kabul edelim. Eleman eklemek istedi imizde bu defa overflow hatasıyla karşılaşacağız demektir. Ayrıca kuyru un solunda bulunan boşluklara da eleman eklenemeyecektir.



ekil 4.4 Basit yapıli kuyrukta meydana gelen problem.

Böyle bir durumdan kurtulmanın yolu, kuyru a dairesel dizi (*circular array*) efekti vermektir. Kuyruk i lemlerinde sürekli ba tan ya da sondan çıkarma veya ekleme yaptı mıza göre elemanlar arasında boşluk yok demektir. Boşluklar ya ba taraftadır, ya da kuyru un son tarafındadır. Kuyruklarda enqueue i leminde önce rear indeksi 1 artırılır ve sonra eleman eklenir. rear kuyru un son indisini gösterdi inde ekleme yapmadan önce 1 arttırıldı na göre indeks dizi boyutuna e itilecek ve ta ma hatası meydana gelecektir. Bunun önüne geçmek için rear indeksi 0 olacak ekleme yapıldığında update edilerek elemanın kuyru un 0 no’lu indisine eklenmesi sa lanır. Peki ama bir kuyrukta 0 no’lu indisin boş oldu unu nereden bilece iz? Bu sorunun cevabı listeler ve yı nılarda oldu u gibi kuyruk dolu mu eklinde bir kontrol yapmalıdır. E er dolu de ilse en azından 0 no’lu indisin boş oldu u kesin olarak bilinmektedir. Gerçekte dairesel bir dizi olmamasına kar ın, böyle bir efekt verilmiş model ekil 4.5’te gösterilmiş tir.



ekil 4.5 Dairesel dizi (*circular array*) modeli.

ekil 4.5’te kuyru un sonundayken dairesel efekt vermek için rear indeksinin de eri 0 yapıyor ve ilk indise eleman ekleniyor. imdi rear 0, front ise 5 indis de erini tutuyor. rear indeksinin front indeksinden daha önde yer

alması uygulanan efektin bir sonucudur. Artık kuyruk modeli biçimlendi ine göre array implementasyonlarındaki yapı tanımlanabilir. Önce kuyruk boyutunu belirleyen sabiti `define` bildirimleriyle yazıyoruz.

```
#define QUEUE_SIZE 10
typedef struct {
    int front; // Hatırlatma, sadece dizi indislerini tutacakları için int türden
    int rear;
    int cnt;
    int data[QUEUE_SIZE];
}queue;
```

Bir Kuyru a Ba langıç De erlerini Vermek (initialize)

queue yapısını kullanmadan önce initialize etmek gerekir. Fonksiyon oldukça basittir ve yapı de i kenlerinin ilk de erlerini vermektedir.

```
void initialize(queue *q) {
    q -> front = 0;
    q -> rear = -1; // Önce arttırılaca ı ve sonra ekleme yapılaca ı için -1
    q -> cnt = 0;
}
```

Kuyru un Bo Olup Olmadı ının Kontrolü (isEmpty)

Yı mlarda ki fonksiyonla aynıdır. Eleman sayısını tutan cnt yapı de i kenini 0 ise kuyruk bo tur ve geriye true döndürülür. E er cnt 0'dan büyükse kuyruksa eleman vardır ve geriye false de eri döndürülür.

```
typedef enum {false, true}boolean;
boolean isEmpty(queue *q) {
    return(q -> cnt == 0);
}
```

Kuyru un Dolu Olup Olmadı ının Kontrolü (isFull)

Yı mlarda ki fonksiyonla aynıdır. Eleman sayısını tutan cnt yapı de i kenini QUEUE_SIZE'a e it ise kuyruk doludur ve geriye true döndürülür. E er cnt QUEUE_SIZE'dan küçükse kuyruksa dolu de ildir ve geriye false de eri döndürülür.

```
boolean isFull(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}
```

Kuyru a Eleman Ekleme (enqueue)

Bir kuyru a eleman ekleme fonksiyonu enqueue olarak bilinir ve bu fonksiyon, kuyru un adresini ve eklenecek elemanın de erini parametre olarak alır. Geri dönü de eri olmayaca ı için türü void olmalıdır. Ayrıca eleman ekleyebilmek için kuyru un dolu olmaması gerekir.

```
void enqueue(queue *q, int x) {
    if(!isFull(q)) {
        // kuyruk dolu mu diye kontrol ediliyor
        q -> rear ++; // ekleme öncesi rear arttırılıyor
        q -> cnt ++;

        if(q -> rear == QUEUE_SIZE)
            q -> rear = 0;
        /* Ekleme yapmadan önce rear 1 arttırılıyordu. Kuyruk dolu olmayabilir fakat
        rear indeksi dizinin son indisini gösteriyor olabilir. Böyle bir durumda
        dizide taşma hatası olmaması için rear'in de eri kontrol ediliyor. rear son
        indisi gösteriyor ve kuyruk dolu de ilse rear de eri 0'a set ediliyor */
        q -> data[q -> rear] = x;
    }
}
```

Eleman eklendikten sonra rear de i kenini en son eklenen elemanın indis de erini tutmaktadır.

Kuyruktan Eleman Çıkarma lemi (dequeue)

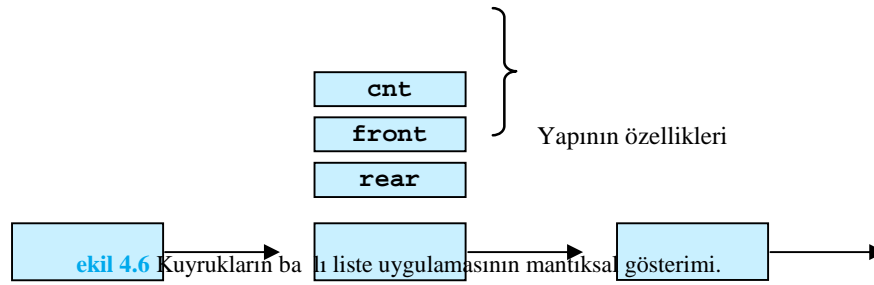
dequeue fonksiyonu olarak bilinir. İlk giren elemanı çıkarmak demektir. Çıkarılan elemanın veri de eriyle geri dönece i için fonksiyonun tipi int olmalıdır.

```
int dequeue(queue *q) {
    if(!isEmpty) { // kuyruk boş mu diye kontrol ediliyor
        int x = q -> data[q -> front]; // front de erinin saklanması gerekiyor
        q -> front++;
        q -> cnt--;

        if(q -> front == QUEUE_SIZE)
            /* front dizinin sonunu gösteriyor olabilir. Dizi taşma hatası olmaması
            için kontrol edilmesi gerekir. */
            q -> front = 0;
        return x; // çıkarılan elemanın data de eriyle ça rıldı ı yere geri dönüyor
    }
}
```

4.3 KUYRUKLARIN BA LI L STE (Linked List) MPLEMENTASYONU

Kuyrukların ba lı liste uygulamasında, üç adet özellikleri bulunmaktadır. Bunlardan cnt eleman sayısını tutan int türden bir yapı de i kenidir. kincisi struct node türden front i aretçisi kuyru un önünü göstermektedir ve üçüncüsü ise yine struct node türden rear i aretçisidir. rear ise kuyru un arkasını göstermektedir. Bu uygulamada eleman ekleme i lemi rear üzerinden, çıkarma i lemi ise front üzerinden yapılır. Bunlar bir kuyrukta ekleme ve çıkartma i lemlerini kolay bir eilde yapabilmek için tanımlanmı tır. Yine ekleme ve çıkarma i lemlerinde kolay eri im için bir de cnt de i kenini tanımlanmı tır.



Veri yapısı a a ıdaki gibi tanımlanmı tır.

```
typedef struct {
    int cnt;
    struct node *front;
    struct node *rear;
}queue;
```

Kuyru a Ba langıç De erlerini Vermek (initialize)

Her zaman oldu u gibi ilk önce initialize yapılması gerekiyor. Fonksiyon u eilde tanımlanabilir;

```
void initialize(queue *q) {
    q -> cnt = 0;
    q -> front = q -> rear = NULL; // NULL de eri her iki işaretçiye de atanır
}
```

initialize fonksiyonunda q -> front = q -> rear = NULL; satırındaki gibi bir i lemde NULL de eri ilk önce q -> rear'a atanır. Bu defa da q -> front göstericisine q -> rear de eri atanarak iki i aretçi de NULL de erini almı olur.

Kuyru un Bo Olup Olmadı ının Kontrolü (isEmpty)

Eleman sayısını tutan cnt yapı de i keni 0 ise kuyruk bo tur ve geriye true döndürülür. E er cnt 0'dan büyükse kuyrukta eleman vardır ve geriye false de eri döndürülür.

```
int isEmpty(queue *q) {
    return(q -> cnt == 0);
}
```

Kuyru un Dolu Olup Olmadı ının Kontrolü (isFull)

Yıllarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı da `i` ken `QUEUE_SIZE`'a `e` ise kuyruk doludur ve geriye `true` döndürülür. Eğer `cnt` `QUEUE_SIZE`'dan küçükse kuyrukte dolu değildir ve geriye `false` de geri döndürülür.

```
int isFull(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}
```

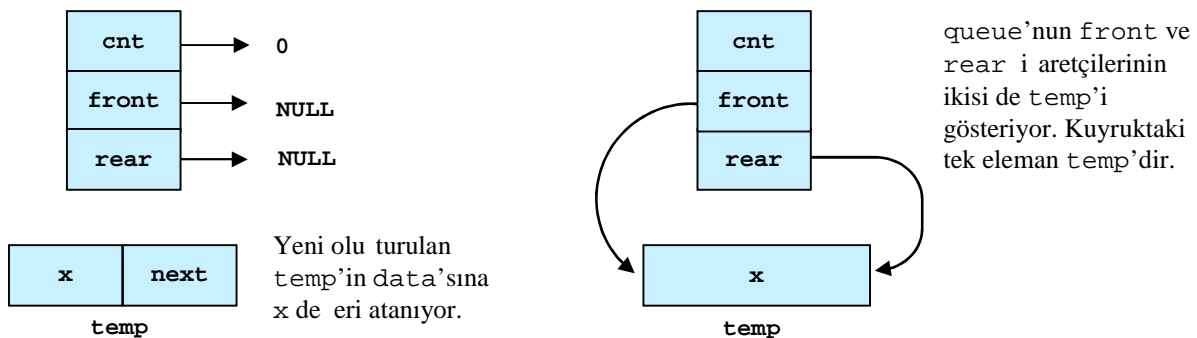
Kuyruğa Eleman Ekleme (enqueue)

Bir kuyruğa eleman ekleme fonksiyonu `enqueue` olarak bilinir ve bu fonksiyon, kuyruğun adresini ve eklenecek elemanın değeri parametre olarak alır. Geri dönüş değeri olmayacağı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için kuyruğun dolu olmaması gerekir. `stack`'lerdeki `push` işlemi gibidir.

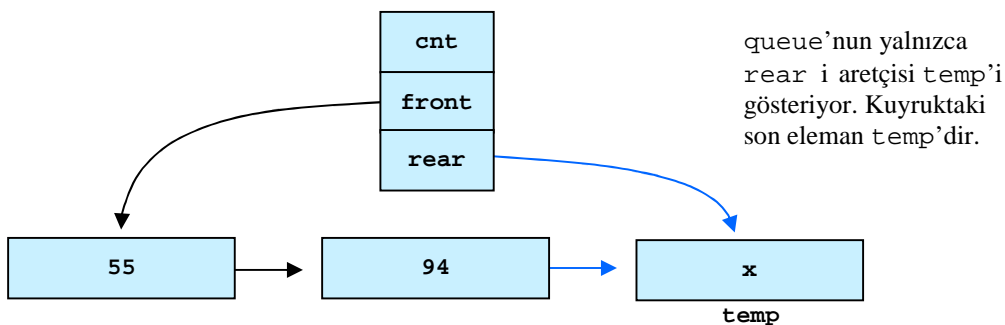
```
void enqueue(queue *q, int x) {
    if(!isFull(q)) {
        // kuyruk dolu mu diye kontrol ediliyor
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        // C++'ta struct node *temp = new node(); şeklinde
        temp -> next = NULL;
        temp -> data = x;

        if(isEmpty(q))
            q -> front = q -> rear = temp;
        else {
            q -> rear -> next = temp;
            q -> rear = temp;
        }
        q -> cnt ++;
    }
}
```

Fonksiyon içerisinde `struct node` türünden `temp` adında bir yapı daha oluşturuyoruz. Öncelikle atamaları `temp`'in elemanlarına yapıp sonra kuyruğun arkasına ekliyoruz. Kuyruk tamamen boş ise `front` ve `rear` işaretçilerinin ikisi de kuyruğa yeni eklenen düğümü gösterecektir. Çünkü ilk eleman da son eleman da eklenen bu yeni düğümdür. Eğer kuyruk boş değilse sadece arkaya ekleme yapıldığına dikkat ediniz. Kuyruğun tamamen boş olması durumu ve en az bir elemanı olması durumuna göre `enqueue` işlemini eklemelerle modelleyelim.



ekil 4.7 a) Boş bir kuyruğa eleman eklenmesi.



ekil 4.7 b) Dolu bir kuyruğa eleman eklenmesi.

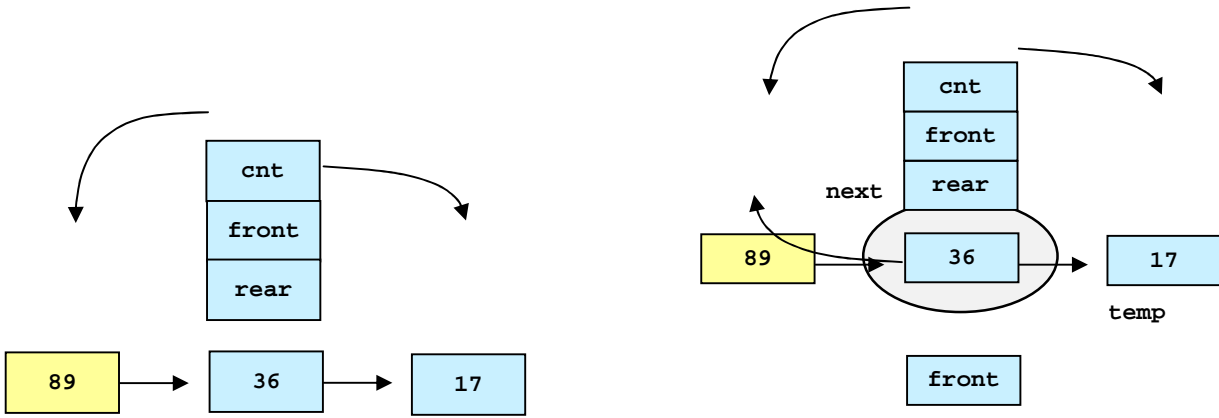
Ekleme yapmadan önce `rear` içinde 94 verisi bulunan düğümü gösteriyordu. İmdi ise `temp`'i gösteriyor. Burada anlaşılması gereken nokta şudur; `queue` yalnızca elemanların başını, sonunu ve sayısını tutan tek bir yapıdır. Eklenen tüm veriler `struct node` türden birer düğümdür.

Kuyruktan Eleman Çıkarma İlemi (dequeue)

Bir kuyruktan eleman çıkarma fonksiyonu dequeue olarak bilinir ve fonksiyon, kuyruğun adresini parametre olarak alır. İlk giren elemanı çıkarmak demektir. Çıkarılan elemanın veri deeriyle geri dönmeye için fonksiyonun tipi int olmalıdır. Eleman çıkartabilmek için kuyruğun boş olmaması gerekir. stack'lerdeki pop işlemi gibidir.

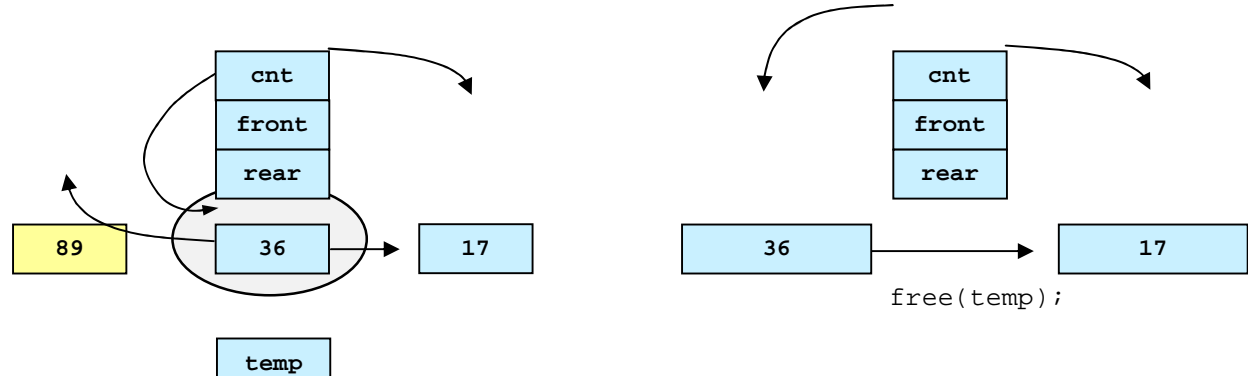
```
int dequeue(queue *q) {
    if(!isEmpty(q)) { // kuyruk boş mu diye kontrol ediliyor
        struct node *temp = q -> front;
        int x = temp -> data; // silmeden önce geri döndürülecek veri saklanıyor
        q -> front = temp -> next;
        free(temp); // Daha önce front'un gösterdiği adres belleğe geri veriliyor
        q -> cnt--;
        return x; // çıkarılan elemanın data deeriyle çağırıldığı yere geri dönüyor
    }
}
```

Kuyruğun front işaretçisinin gösterdiği adres temp'e atanmıştır. İmdi aynı düümün adresi hem front işaretçisinde hem de temp işaretçindedir. Şekilde dequeue işlemi açık bir şekilde gösterilmiştir.



ekil 4.8 a) Kuyruktan çıkarılacak eleman sarı renkli olarak görülüyor.

ekil 4.8 b) İmdi temp de silinecek elemanı gösteriyor.



Silinecek düümün next işaretçisi bir sonraki düümün adresini tutmaktadır ($front \rightarrow next$, ve $temp \rightarrow next$ de aynı adresi tutuyor demektir). Şu anda hem front'un hem de temp'in data'sı olan 89 deeri kenine aktarıyor. Sonra $temp \rightarrow next$ deeri front işaretçisine atanarak 36 verisinin bulunduğu düümün göstermesi sağlanıyor. Nihayet temp işaretçisinin gösterdiği adres $free()$ fonksiyonuyla belleğe geri kazandırılıyor ve elemanın sayısını tutan cnt deeri de 1 azaltılarak dequeue işlemi tamamlanıyor.

Örnek 4.1: Palindrom, tersten okundu da aynı olan cümle, sözcük ve sayılara denilmektedir (*Ey Edip, Adana'da pide ye, 784521125487 ...vb*). Verilen bir stringin palindrom olup olmadığını belirleyen C kodunu, noktalama işaretleri, büyük harfler ve boşlukların ihmal edilmesini varsayarak stack ve queue yapılarıyla yazalım.

ANSI C'de;

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



```

#include <conio.h>
#include <ctype.h>
#define SIZE 100
#define STACK_SIZE 100
#define QUEUE_SIZE 100

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

typedef struct {
    struct node *top;
    int cnt;
}stack;

typedef struct {
    int cnt;
    struct node *front;
    struct node *rear;
}queue;

void initialize_stack(stack *stk) {
    stk -> top = NULL;
    stk -> cnt = 0;
}

void initialize_queue(queue *q) {
    q -> cnt = 0;
    q -> front = q -> rear = NULL;
}

typedef enum {false, true}boolean;
boolean isEmpty_stack(stack *stk) {
    return(stk -> cnt == 0);
}

boolean isFull_stack(stack *stk) {
    return(stk -> cnt == STACK_SIZE);
}

void push(stack *stk, int c) {
    if(!isFull_stack(stk)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));

        temp -> data = c;
        temp -> next = stk -> top;
        stk -> top = temp;
        stk -> cnt++;
    }
    else
        printf("Stack dolu\n");
}

int pop(stack *stk) {
    if(!isEmpty_stack(stk)) {
        struct node *temp = stk -> top;
        stk -> top = stk -> top -> next;
        int x = temp -> data;
        free(temp);
        stk -> cnt--;
        return x;
    }
}

int isEmpty_queue(queue *q) {
    return(q -> cnt == 0);
}

```

```

int isFull_queue(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}

void enqueue(queue *q, int x) {
    if(!isFull_queue(q)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> next = NULL;
        temp -> data = x;

        if(isEmpty_queue(q))
            q -> front = q -> rear = temp;
        else
            q -> rear = q -> rear -> next = temp;
        q -> cnt ++;
    }
}

int dequeue(queue *q) {
    if(!isEmpty_queue(q)) {
        struct node *temp = q -> front;
        int x = temp -> data;
        q -> front = temp -> next;
        free(temp);
        q -> cnt--;
        return x;
    }
}

int main() {
    char ifade[SIZE];
    queue q;
    stack s;
    int i = 0, mismatches = 0;

    initialize_stack(&s);
    initialize_queue(&q);

    printf("Bir ifade giriniz...\n");
    gets(ifade);

    for(i = 0; i != strlen(ifade); i++) {
        if(isalpha(ifade[i])) {
            enqueue(&q, tolower(ifade[i]));
            push(&s, tolower(ifade[i]));
        }
    }
    while(!isEmpty_queue(&q)) {
        if(pop(&s) != dequeue(&q)) {
            mismatches = 1;
            break;
        }
    }

    if(mismatches == 1)
        printf("Girdiginiz ifade palindrom degildir!\n");
    else
        printf("Girdiginiz ifade bir palindromdur!\n");

    getch();
    return 0;
}

```

C++'ta ise birkaç farklılık dışında bir şey yoktur.

```

#include <cstdlib>
#include <string>
#include <iostream>

```

```

#include <conio.h>
#include <ctype.h>
#define SIZE 100
#define STACK_SIZE 100
#define QUEUE_SIZE 100
using namespace std;

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

typedef struct {
    struct node *top;
    int cnt;
}stack;

typedef struct {
    int cnt;
    struct node *front, *rear;
}queue;

void initialize_stack(stack *stk) {
    stk -> top = NULL;
    stk -> cnt = 0;
}

void initialize_queue(queue *q) {
    q -> cnt = 0;
    q -> front = q -> rear = NULL;
}

bool isEmpty_stack(stack *stk) {
    return(stk -> cnt == 0);
}

bool isFull_stack(stack *stk) {
    return(stk -> cnt == STACK_SIZE);
}

void push(stack *stk, int c) {
    if(!isFull_stack(stk)) {
        struct node *temp = (struct node *)malloc(sizeof(struct node));
        temp -> data = c;
        temp -> next = stk -> top;
        stk -> top = temp;
        stk -> cnt++;
    }
    else
        printf("Stack dolu\n");
}

int pop(stack *stk) {
    if(!isEmpty_stack(stk)) {
        struct node *temp = stk -> top;
        stk -> top = stk -> top -> next;
        int x = temp -> data;
        free(temp);
        stk -> cnt--;
        return x;
    }
}

int isEmpty_queue(queue *q) {
    return(q -> cnt == 0);
}

int isFull_queue(queue *q) {

```

```

    return(q -> cnt == QUEUE_SIZE);
}

void enqueue(queue *q, int x) {
    if(!isFull_queue(q)) {
        struct node *temp = new node();
        temp -> next = NULL;
        temp -> data = x;

        if(isEmpty_queue(q))
            q -> front = q -> rear = temp;
        else
            q -> rear = q -> rear -> next = temp;
        q -> cnt ++;
    }
}

int dequeue(queue *q) {
    if(!isEmpty_queue(q)) {
        struct node *temp = q -> front;
        int x = temp -> data;
        q -> front = temp -> next;
        free(temp);
        q -> cnt--;
        return x;
    }
}

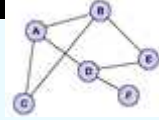
int main() {
    string the_string;
    queue q;
    stack s;

    initialize_stack(&s);
    initialize_queue(&q);
    int i = 0, mismatches = 0;

    cout << "Bir ifade giriniz" << endl;
    cin >> the_string;

    while(i < the_string.length()) {
        if(isalpha(the_string[i])) {
            enqueue(&q, tolower(the_string[i]));
            push(&s, tolower(the_string[i]));
        }
        i++;
    }
    while(!isEmpty_queue(&q)) {
        if(pop(&s) != dequeue(&q)) {
            mismatches = 1;
            break;
        }
    }
    if(mismatches == 1)
        cout << "Girdiginiz ifade palindrom degildir!" << endl;
    else
        cout << "Girdiginiz ifade bir palindromdur!" << endl;
    getch();
    return EXIT_SUCCESS;
}

```



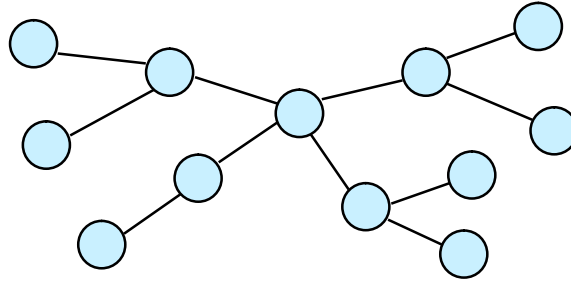
BÖLÜM

A ağlar (Trees) 5

5.1 GR

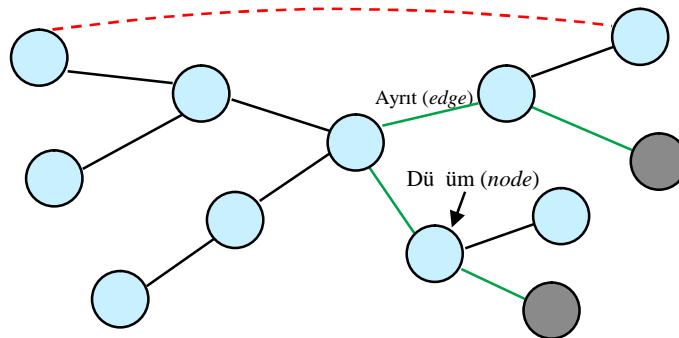
A ağ, verilerin birbirine sanki bir ağ yapısı oluyormuş gibi sanal olarak bağlanmasıyla elde edilen hiyerarşik yapıya sahip bir veri modelidir; bilgisayar yazılım dünyasında, birçok yerde / uygulamada programcının karşısına çıkar. Ağ veri yapılarının işletim sistemlerinin dosya sisteminde, oyunların olası hamlelerinde ve şirketlerdeki organizasyon yapısı vb. gibi birçok uygulama alanları vardır. Örneğin, NTFS dosya sistemi hiyerarşik dosyalama sistemini kullanır. Yani bu sistemde bir kök dizin ve bu kök dizine ait alt dizinler bulunur. Bu yapı **parent-child** ilişkisi olarak da adlandırılır. Windows gezgininde dosyaların gösterilmesi esnasında ağ veri yapısı kullanılır. Bunun sebebi üstlük altlık ilişkileridir. Arama algoritmalarında (*ikili arama – binary search*) kullanılabilir. Daha detaylı bir örnek verecek olursak, bir satranç tahtası üzerinde atın 64 hamlede tüm tahtayı dolaşması problemi ağ veri yapısıyla çözülebilen bir problemdir. Birçok problemin çözümü veya modellenmesi, doğası gereği ağ veri modeline çok uygundur. Ağ veri modeli daha fazla bellek alanına gereksinim duyar. Çünkü ağ veri modelini kurmak için birden çok ilişkiyi kullanılır. Buna karşın, yürütme zamanında sağladığı getiri ve ağ üzerinde işlem yapacak fonksiyonların rekürsif yapıda kolayca tasarlanması ve kodlanması ağ veri modelini uygulamada ciddi bir seçim yapmaktadır.

Ağ veri yapısı çizge (*graph*) veri yapısının bir alt kümesidir. Bir ağ, düğümler (*node*) ve bu düğümleri birbirine bağlayan ayrıtlar (*edge–dal-kenar*) kümesine **ağ** (*tree*) denir. Her düğüm ve ayrıtları olan küme ağ değildir. Bir çizgenin ağ olabilmesi için her iki düğüm arasında sadece bir **yol** olmalı, devre (*cycle, çevrim*) olmamalıdır. **Yol** (*path*) birbirleri ile bağlantılı ayrıtlar (*edge*) dizisidir.



ekil 5.1 Bir ağ yapısının görünümü.

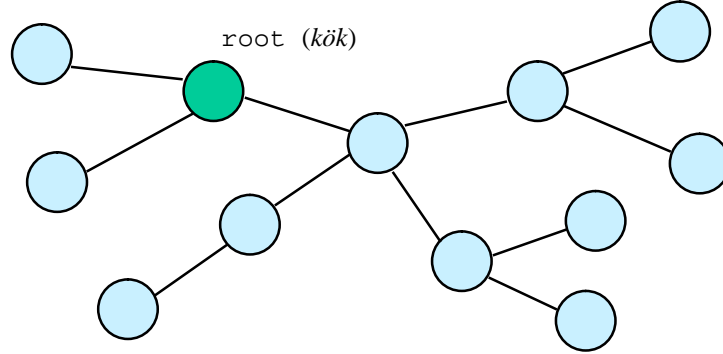
ekil 5.1’de görülen yapı bir ağdır ve mutlaka bir ağda kök bulunmak zorunda değildir. Herhangi iki düğüm arasında yalnızca bir yol vardır ve bir çevrim içermemektedir.



ekil 5.2 Bir ağda herhangi iki düğüm arasında yalnızca bir yol bulunur (yani ayrıtların olduğu yol).

ekil 5.2'de koyu gri renkte gösterilmi olan dü ümler arasında yalnızca bir yol bulunmaktadır. Eğer kesikli kırmızı çizgi ile belirtilen bir çevrim (*cycle*) daha olsaydı, bu bir ağ değil, sadece graf olacaktı. Çünkü dü ümlere ikinci bir yol ile de erişilebilme imkânı ortaya çıkmaktadır.

Kökü Olan Ağ (Rooted Tree): Kökü olan ağta özel olan ya da farklı olan dü üm kök (*root*) olarak adlandırılır. Kök hariç her *c* (*child*) dü ümünün bir *p* ebeveyni (*parent*) vardır. Alılgelimi ekliyle ağlarda, kök en yukarıda yer alıp çocuklar alt tarafta olacak biçimde çizilir. Fakat böyle gösterme zorunlulu u yoktur.



ekil 5.3 Rooted tree modeli.

Ebeveyn (Parent): Bir *c* dü ümünden köke olan yol üzerindeki ilk dü ümdür. Bu *c* dü ümü *p*'nin çocuğudur (*child*).

Yaprak (Leaf): Çocuğ u olmayan dü ümdür.

Karde (Sibling): Ebeveyni aynı olan dü ümlerdir.

Ata (Ancestor): Bir *d* dü ümünün ataları, *d*'den köke olan yol üzerindeki tüm dü ümlerdir.

Descendant: Bir dü ümün çocukları, torunları vs. gibi sonraki neslidir.

Yol Uzunlu u: Yol üzerindeki ayrıt sayısıdır.

***n* Dü ümünün Derinli i:** Bir *n* dü ümünden köke olan yolun uzunluğudur. Kökün derinli i sıfırdır.

***n* Dü ümünün Yüksekli i:** Bir *n* dü ümünden en alttaki descendant dü ümüne olan yolun uzunluğudur. Başka bir deyişle neslinin en sonu olan dü ümdür.

Bir Ağacın Yüksekli i: Kökün yüksekliğine ağacın yüksekli i de denir.

***n* Dü ümünde Köklenen Alt Ağ (Subtree Rooted at *n*):** Bir *n* dü ümü ve *n* dü ümünün soyu (*descendant*) tarafından oluşan ağdır.

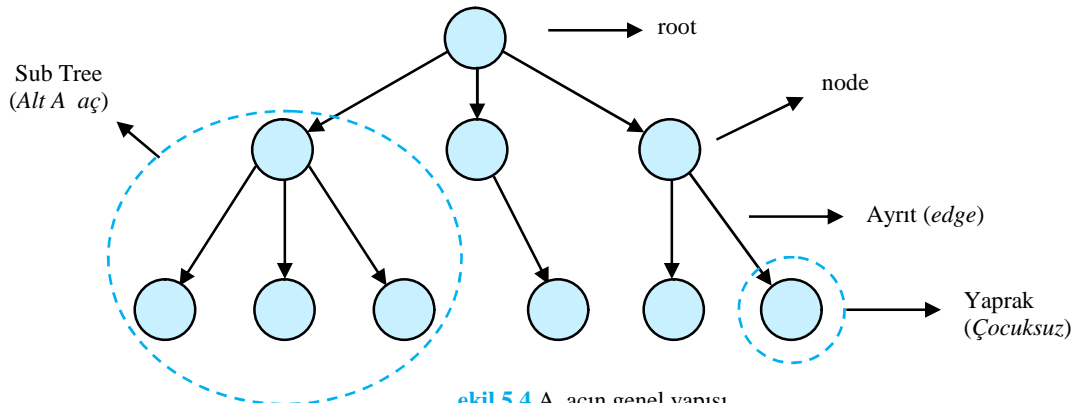
Binary Tree (kili Ağ): kili ağta bir dü ümün sol çocuk ve sağ çocuk olmak üzere en fazla iki çocuğ u olabilir.

Düzey (level) / Derinlik (depth): Kök ile dü üm arasındaki yolun üzerinde bulunan ayrıtların sayısıdır. Bir dü ümün kök dü ümden olan uzaklığıdır. Kökün düzeyi sıfırdır (*bazı yayınlarda 1 olarak da belirtilmektedir*).

5.2 A AÇLARIN TEMSİLİ

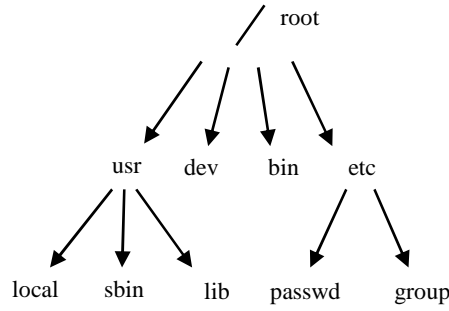
Genel ağ yapısında dü ümlerdeki çocuk sayısında ve ağ yapısında bir kısıtlama yoktur. Ağ yapısına belli kısıtlamalar getirilmesiyle ağ türleri meydana gelmiştir. Her yaprağın derinli i arasındaki fark belli bir sayıdan fazla (*örneğin 2*) olmayan ağlara **dengeleli ağlar** (*balanced trees*) denir. kili ağlar (*binary trees*) ise dü ümlerinde en fazla iki bağ içeren (0, 1 veya 2) ağlardır. Ağ yapısı kısıtlamaların az olduğu ve problemin kolaylıkla uyarlanabileceği bir yapı olduğundan birçok alanda kullanılmaktadır. Örnek olarak işletim sistemlerinde kullandığımız dosya-dizin yapısı tipik bir ağ modellemesidir.

Ağ veri modelinde, bir köki üretçisi, sonlu sayıda dü ümleri ve onları birbirine bağlayan dalları vardır. Veri ağacının dü ümlerinde tutulur. Dallarda ise geçi ko ulları vardır. Her ağacın bir köki üretçisi bulunmaktadır. Ağaca henüz bir dü üm eklenmemiş ise ağ boşdur ve köki üretçisi NULL değeri gösterir. Ağ bu kök etrafında dallanır ve genişler.



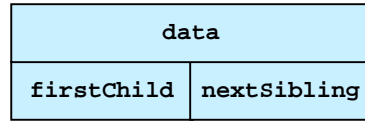
ekil 5.4 Ağacın genel yapısı.

1) **Tüm Aaçlar için:** Unix işletim sisteminde bir dosya hiyerarşisi vardır. Bu dosya sisteminde en üstte kök dizin (*root*) bulunur. Sistemdeki tüm diğer dosya ve dizinler bunun altında toplanırlar. Ters dönmü bir aaç gibidir.



ekil 5.5 UNIX dosyalama hiyerarşisi modeli.

ekil 5.5'de gösterilen dosya hiyerarşisindeki aaç ikili bir aaç değildir. Çünkü bazı düğümlerin üç çocuğu, kökün ise dört çocuğu bulunmaktadır. Binary aaçlarda bir sol çocuk, bir de sağ çocuk bulunmaktaydı. Oysa bu aaç türünde diğer çocuklar da yer alıyor. ekil 5.5'de görüldüğü gibi *root*'ün çocukları olan *usr*'yi *left* ile gösterdi imizi, *dev*'i de *right* ile gösterdi imizi düğümlerim. Peki *bin* ile *etc*'yi nasıl gösterebiliriz? Bu durumda farklı bir veri yapısı tanımlamamız gerekecektir. Bu yapının ekli 5.6'da gösterilmiştir.



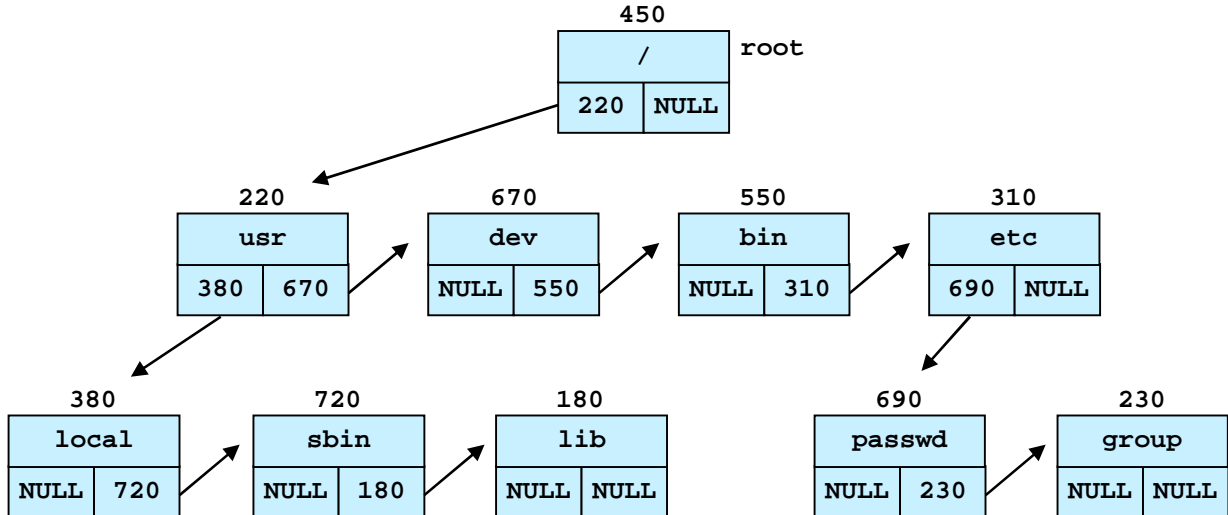
ekil 5.6 ikili olmayan aaçlarda bir node'un yapısı.

Kök düğümün kardeşi yoktur, sadece çocukları ve torunları olabilir. ikili olmayan aaçların gösteriminde her ebeveyn yalnızca ilk çocuğunu göstermeli ve eğer varsa kendi kardeşi de göstermelidir. Kardeşler birden fazla olabilir. Öyleyse ilk çocuktan sonra kardeşler birbirlerine bağlanmalıdır. Bu açıklamalara uygun veri yapısı aaçtaki gibi yazılabilir. Burada da yine ihtiyaç halinde bir `struct node *parent` göstericisi tanımlanabilir.

```

struct node {
    int data;
    struct node *firstChild; // ilk çocuk
    struct node *nextSibling; // kardeş
};
  
```

Kök düğümün kardeşi olmayacağı için yalnızca `firstChild` bağlantısı var gibi görülse de diğer kardeşlerin düğüme `firstChild` işaretçisinin gösterdiği ilk çocuk düğümünün `nextSibling` işaretçisinden erişilebilir. **Bu yöntem bellek alanından kazanç sağlar;** ancak, programın yürütme zamanını artırır. Çünkü bir düğümden onun çocuklarına doğrudan erişim ortadan kalkmış, bağlı listelerde olduğu gibi ardışık erişime ortaya çıkmıştır. Aaçta, ekil 5.5'deki UNIX dosya yapısının mantıksal modellenmesi gösterilmiştir.

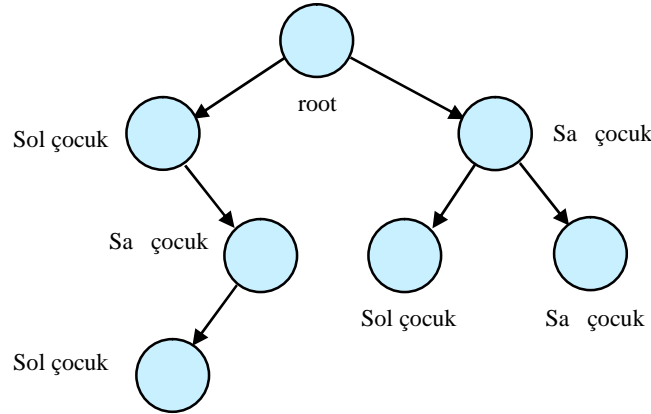


ekil 5.7 UNIX işletim sistemindeki dosya yapısının veri modeli.

2) kili A ağlar (Binary Trees) çin: E er bir a açtaki her dü ümün en fazla 2 çocu u varsa bu a aca ikili a aç denir. Di er bir deyi le bir a açtaki her dü ümün derecesi en fazla 3 ise o a aca ikili a aç denir. kili a ağların önemli özelli i e er dengeli ise çok hızlı bir ekilde ekleme, silme ve arama i lemlerini yapabilmesidir. A aç veri yapısı üzerinde i lem yapan fonksiyonların birço u kendi kendini ça ıran (*recursive*) fonksiyonlardır. Çünkü bir a acı alt a ağlara böldü ümüzde elde edilen her parça yine bir a aç olacaktır ve fonksiyonumuz yine bu a aç üzerinde i lem yapaca ı için kendi kendini ça ıracaktır.

kili a aç veri yapısının tanımında a aç veri yapısının tüm tanımları geçerlidir, farklı olan iki nokta ise unlardır:

- A acın derecesi (*degree of tree*) 2'dir.
- Sol ve sa alt a acın yer de i tirmesiyle çizilen a aç aynı a aç de ildir.



ekil 5.8 kili a ağların gösterimi.

kili a aç (*binary tree*) veri yapısının yo un biçimde kullanılır olu u iki önemli uygulama alanı ile ili kilidir. Sıralama algoritmaları ile derleyicilerde sözdizim çözümlleme (*syntax analysis*), kod geli tirme (*code optimization*) ve amaç kod üretme (*object code generation*) algoritmaları bu veri yapısını kullanırlar. Ayrıca kodlama kuramı (*coding theory*), turnuva düzenleme, aile a acı gösterimi ve benzerleri tekil kullanımlar da söz konusudur.

A ağlarla ilgili özellikler:

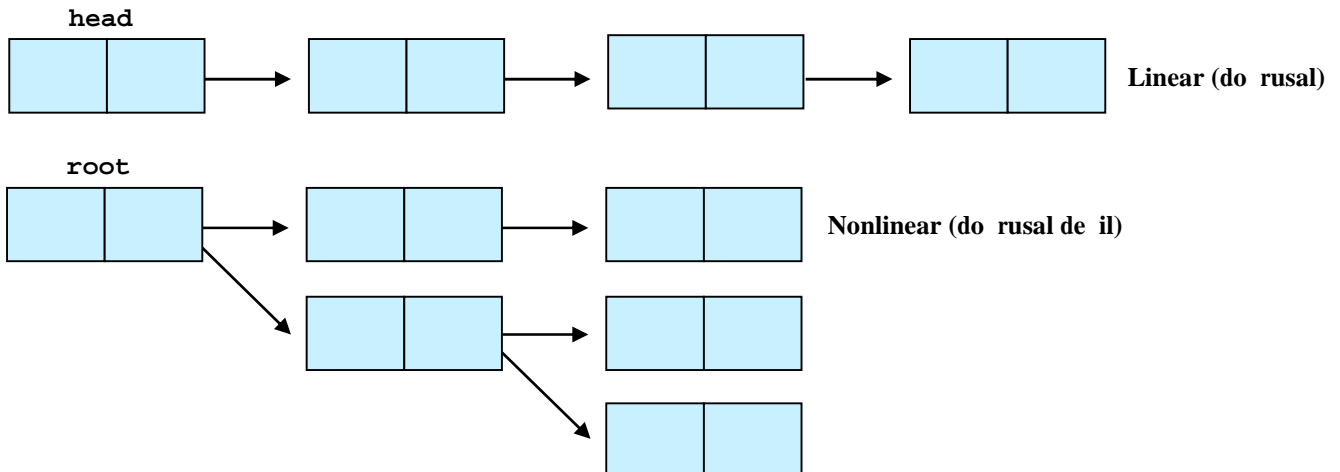
- Bir binary a açta, bir i seviyesindeki maksimum node sayısı 2^i 'dir. ($i, 0$ oldu u için).
- K derinli ine sahip bir binary a açta maksimum node sayısı $2^{K+1} - 1$ 'dir.
- Seviye ve derinlik kökle ba layacak ekilde aynı sayılmaktadır.
- A açta node'ların eksik kısımları NULL olarak adlandırılır.

kili bir a aç için veri yapısı a a ıdaki gibi tanımlanabilir;

```

struct node {
    int data;
    struct node *left;
    struct node *right;
};
  
```

htiyaç oldu unda yapı içerisinde bir de parent tanımlanabilir. A ağların yapısının da ba lı liste oldu u tanımdan anla ılabilir. Fakat normal ba lı listelerden farkı **nonlinear** olmasıdır. Yani do rusal de ildir.



ekil 5.9 A ağlarla ba lı listelerin kar ıla tırılması.

kili A ağlar Üzerinde Dola ma

kili a ağ üzerinde dola ma birçok ekilde yapılabilir Ancak, rastgele dola mak yerine, önceden belirlenmi bir yömeme, bir kurala uyulması algoritmik ifadeyi kolayla tırır. Üstelik rekürsif fonksiyon yapısı kullanılırsa a ağ üzerinde i lem yapan algoritmaların tasarımı kolayla ır. Önce-kök (*preorder*), kök-ortada (*inorder*), sonra-kök (*postorder*) olarak adlandırılan üç de i ik dola ma ekli çe itli uygulamalara çözümler olmaktadır.

1- Preorder (Önce Kök) Dola ma: Önce kök yakla ımında ilk olarak *root* (*kök*), sonra *left* (*sol alt a ağ*) ve ardından *right* (*sa alt a ağ*) dola ılır. Fonksiyonu tanımlamadan önce kolayca kullanabilmek için `typedef` bildirimleriyle bir yapı nesnesi olu turulabilir. Bildirim `typedef struct node *BTREE;` ekinde asterisk (*) konularak da yapılabilir. Böylece her BTREE türünde geriye adres döndüren fonksiyonların ve adres de i kenlerinin bildiriminde asterisk i aretinden kurtulunmu olunur fakat bazı derleyicilerin kararsız çalı masına neden olabilmektedir.

```
typedef struct node BTREE;
```

BTREE yapısı `struct node` veri yapısıyla tanımlanmı olan bir çe it `node`'dur. çerisinde veri tutan `data` de i ken i, adres bilgisi tutan `left` ve `right` isimli iki i aretçisi bulunmaktadır. kili a ağta dola ırken fonksiyon geriye herhangi bir ey döndürmeyece i ve sadece a ağ üzerinde dola aca ı için türü `void` olmalıdır.

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

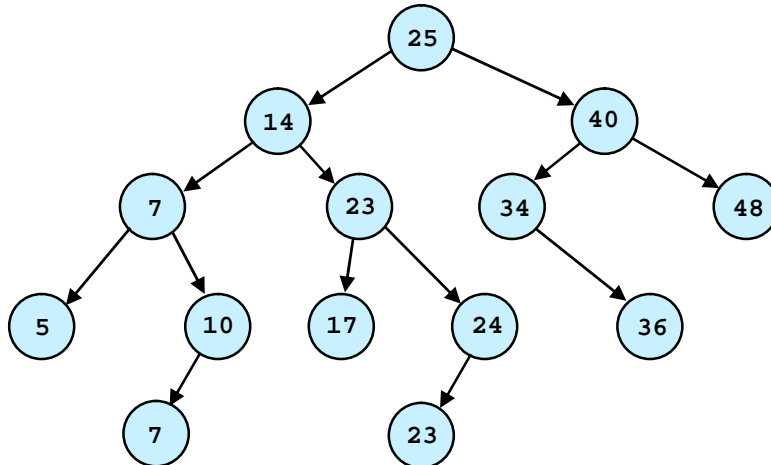
2- Inorder (Kök Ortada) Dola ma: Inorder dola mada önce *left* (*sol alt a ağ*), sonra *root* (*kök*) ve *right* (*sa alt a ağ*) dola ılır. Fonksiyonun türü yine `void` olmalıdır.

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

3- Postorder (Kök Sonda) Dola ma: Postorder yakla ımında ise, önce *left* (*sol alt a ağ*), sonra *right* (*sa alt a ağ*) ve *root* (*kök*) dola ılır. Fonksiyon `void` türündedir.

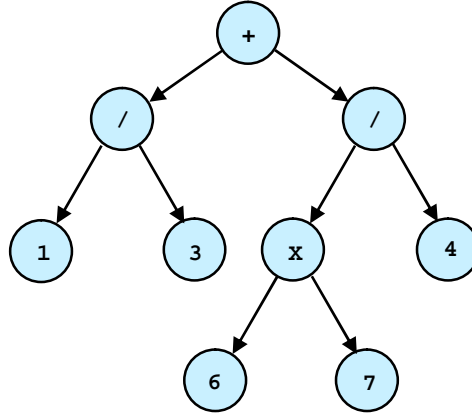
```
void postorder(BTREE *root) {
    if(root != NULL) {
        postorder(root -> left);
        postorder(root -> right);
        printf("%d", root -> data);
    }
}
```

Örnek 5.1: “25, 14, 23, 40, 24, 23, 48, 7, 5, 34, 10, 7, 17, 36” de erlerine sahip dü ümler için ikili a ağ gösterimini olu turunuz ve üç farklı *preorder*, *inorder* ve *postorder* sıralama yöntemine göre yazınız.



Çözüm: Preorder : 25-14-7-5-10-7-23-17-24-23-40-34-36-48
 Inorder : 5-7-7-10-14-17-23-23-24-25-34-36-40-48
 Postorder : 5-7-10-7-17-23-24-23-14-36-34-48-40-25

Örnek 5.2: A a da görülen ba ntı a acındaki verileri preorder, inorder ve postorder sıralama yöntemine göre yazınız.



Çözüm: Preorder : + / 1 3 / x 6 7 4
 Inorder : 1 / 3 + 6 x 7 / 4
 Postorder : 1 3 / 6 7 x 4 / +

kili A aç Olu turmak

```

typedef struct node BTREE;

BTREE *new_node(int data) {
    BTREE *p = (BTREE*) malloc(sizeof(BTREE));
    // BTREE *p = new node(); // C++'ta bu şekilde

    p -> data = data;
    p -> left = NULL;
    p -> right = NULL;

    return p;
}
  
```

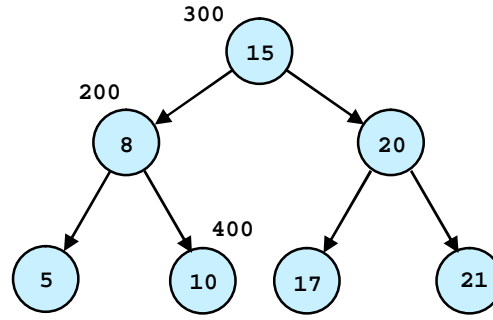
kili A aca Veri Ekleme

kili bir a aca veri ekleme i leminde sol çocu un verisi (*data*) ebeveyninin verisinden küçük olmalı, sa çocu un verisi ise ebeveyninin verisinden büyük ya da e it olmalıdır. Altta insert isimli ekleme fonksiyonunun nasıl yazıldı ını görüyorsunuz. Fonksiyon, veri eklendikten sonra a acın kök adresiyle geri dönmektedir.

```

/* İkili a aca veri ekleyen fonksiyon */
BTREE *insert(BTREE *root, int data) {
    // Fonksiyona gönderilen adresteki a aca ekleme yapılacak
    if(root != NULL) { // a aç boş de ilse
        if(data < root -> data) // eklenecek veri root'un data'sından küçükse
            root -> left = insert(root -> left, data);
        // eklenecek veri root'un data'sından büyük ya da eşitse
        else
            root -> right = insert(root -> right, data);
    }
    else // e er a aç boş ise
        root = new_node(data);
    return root;
}
  
```

Fonksiyonun nasıl çalış tı ını bir ekleme ile açıklayalım. ekil 5.10'da görülen a açta bazı dü ümlerin adresleri üzerlerinde belirtilmi tir.



ekil 5.10: Veri eklenecek örnek bir a aç.

A aca 13 sayısının eklenecek oldu unu kabul edelim. Fonksiyon `insert(300, 13)`; kodu ile ça rılacaktır.

```

if(root != NULL) { // 300 adresinde bir a aç var, yani NULL de il ve koşul do ru
  if(data < root -> data) // 13 root'un data'sı olan 15'ten küçük ve koşul do ru
    root -> left = insert(root -> left, data); // yürütülecek satır
  else
    root -> right = insert(root -> right, data);
}
else
  root = new_node(data);
return root;

```

13, kökün de eri olan 15'ten küçük oldu u için sol çocuk olan 8 verisinin bulundu u 200 adresiyle tekrar ça rılıyor.

```

insert(200, 13);
if(root != NULL) { // 200 adresi NULL de il ve koşul do ru
  if(data < root -> data) // 13, 8'den küçük mü, de il ve koşul yanlış
    root -> left = insert(root -> left, data);
  else // 13, 8'den büyük mü, evet ve koşul do ru
    root -> right = insert(root -> right, data); // yürütülecek satır
}
else
  root = new_node(data);
return root;

```

Bu defa 13, 8'den büyük oldu undan fonksiyon 8'in sa çocu u olan 10 verisinin bulundu u 400 adresiyle ça rılıyor.

```

insert(400, 13);
if(root != NULL) { // 400 adresi NULL de il ve koşul do ru
  if(data < root -> data) // 13, 10'dan küçük mü, de il ve koşul yanlış
    root -> left = insert(root -> left, data);
  else // 13, 10'dan büyük mü, evet ve koşul do ru
    root -> right = insert(root -> right, data); // yürütülecek satır
}
else
  root = new_node(data);
return root;

```

imdi de fonksiyon 10'un sa çocu u ile tekrar ça rılıyor. Fakat `root->right` (sa çocuk) dü üümü henüz olmadı ı için adres ile de il NULL ile fonksiyon ça rılacaktır.

```

insert(NULL, 13);
else // e er a aç boş ise yani dü üüm yok ise
  root = new_node(data); // dü üüm oluşturuluyor ve 13 ekleniyor
return root; // oluşturulan ve veri eklenen dü üümün adresi geri döndürülüyor

```

400 adresindeki 10 datasını bulduran dü üümün sa çocu u olan `right`, NULL de ere sahiptir. Yani herhangi bir dü üümü göstermemektedir, çocukları yoktur. Fonksiyonun en sonunda geri döndürülen yeni dü üümün adresi, 10'un sa çocu unu gösterecek olan ve NULL de ere sahip `right` i aretçisine atanıyor. Sonra fonksiyon kendisinden önceki ça rıldı ı noktaya geri dönüyor. Bölüm 3'te stack konusu ve rekürsif fonksiyonların çalı ma prensibi anlatıldı ı. Fonksiyon, her geri dönü te i leyi anındaki `root`'un adresini geri döndürmektedir.

Örnek 5.3: Klavyeden – 1 girilinceye kadar ağaca ekleme yapan programın kodunu yazınız.

```
#include <stdio.h>
#include <conio.h>
struct node {
    int data;
    struct node *left;
    struct node *right;
};
typedef struct node BTREE;
BTREE *new_node(int data) {
    BTREE *p = (BTREE*) malloc(sizeof(BTREE));
    p -> data = data;
    p -> left = NULL;
    p -> right = NULL;
    return p;
}
BTREE *insert(BTREE *root, int data) { // root'u verilmiş ağaca ekleme yapılacak
    if(root != NULL) {
        if(data < root -> data)
            root -> left = insert(root -> left, data);
        else
            root -> right = insert(root -> right, data);
    }else
        root = new_node(data);
    return root;
}
void preorder(BTREE *root) {
    if(root != NULL) {
        printf("%3d ", root -> data);
        preorder(root -> left);
        preorder(root -> right);
    }
}
void inorder(BTREE *root) {
    if(root != NULL) {
        inorder(root -> left);
        printf("%3d ", root -> data);
        inorder(root -> right);
    }
}
void postorder(BTREE *root) {
    if(root != NULL) {
        postorder(root -> left);
        postorder(root -> right);
        printf("%3d ", root -> data);
    }
}
int main() {
    BTREE *myroot = NULL;
    int i = 0;
    do {
        printf("\n\nAgaca veri eklemek icin sayi giriniz...\nSayi = ");
        scanf("%d", &i);
        if(i != -1)
            myroot = insert(myroot, i);
    } while(i != -1);
    preorder(myroot);
    printf("\n");
    inorder(myroot);
    printf("\n");
    postorder(myroot);
    getch();
    return 0;
}
```

Örnek 5.4: kili bir ağacın sol çocukları ile sağ çocuklarının yerlerini değiştiren `mirror` isimli fonksiyonu yazınız.

Çözüm: Ağacın çocuklarının yerlerinin düzgün bir biçimde değiştirilebilmesi için yapraklarının `parent`'ına kadar inmeli ve `swap` işlemi ondan sonra yapılmalıdır.

```
void mirror(BTREE* root) {
    if(root == NULL)
        return;
    else {
        BTREE* temp;
        mirror(root -> left);
        mirror(root -> right);
        temp = root -> left; // swap işlemi yapılıyor
        root -> left = root -> right;
        root -> right = temp;
    }
}
```

5.3 KİLİ ARAMA AĞAÇLARI (BSTs - Binary Search Trees)

Kili ağaç yapısının önemli bir özelliği de düğümlerin yerleştirilmesi sırasında uygulanan işlemlerdir. Kili bir ağaçta çocuklar ve ebeveyn arasında büyüklük ya da küçüklük gibi bir ilişki yoktur. Her çocuk ebeveyninden küçük veya büyük ya da ebeveynine eşit olabilir. Kili arama ağaçlarındaki (**BST - Binary Search Tree**) durum ise farklıdır. Bir ikili arama ağacındaki her düğüm, sol alt ağacındaki tüm düğümlerden büyük, sağ alt ağacındaki tüm düğümlerden küçük ya da eşittir. Kili arama ağaçlarında her bir düğümün alt ağacı yine bir ikili arama ağacıdır. İnorder sıralama yapıldığında küçükten büyüğe veriler elde edilir.

Kili arama ağaçlarında (*BST*) aynı düğümlere sahip düğümlerin eklenip eklenmeyeceği sıkça sorulan bir sorudur. Bazı kaynaklarda eklenmesinin veri tekrarı açısından uygun olmayacağı belirtilmektedir. Ayrıca boş bir BST ağacına 7 defa 10 değerine sahip düğüm eklendiğini varsayarsak ağacın yüksekliğinin 6 olmasını gerektirir. Bu da ağacın dengesiz olmasına neden olur. Bununla birlikte BST ağaçları zaten dengeyi korumamaktadır. Bunun için AVL ve Kırmızı-Siyah ağaçlar gibi veri yapıları önerilmektedir.

Verinin bütünlüğünün korunması açısından ise aynı düğümler eklenmelidir. Burada ise böyle bir sorun ortaya çıkmaktadır. Tekrarlı düğümlere sahip bir BST ağacından tekrarı olan bir düğümü silmek istediğimizde hangisini sileceğiz? Ya da arama yapmak istediğimizde hangisini bulacağız?

Introduction to Algorithms, 3. baskı kitabında bir BST ağacındaki eşit düğümlere sahip düğümleri bir bağlı liste ile gösterilmesi önerilmektedir. Ancak bu da veri yapısında fazladan bir alan demektir.

Biz bu derste ağacındaki algoritmada gösterildiği üzere eşit düğümleri ağacın sağ tarafına ekleyeceğiz.

Kili Arama Ağacına Veri Eklemek

Kili bir arama ağacında sol çocuğun verisi ebeveyninin datasından küçük olmalı ve sağ çocuğun verisi de ebeveyninin datasından büyük ya da eşit olmalıdır. Altta `insertBST` isimli ekleme fonksiyonunun kodu görülüyor. Fonksiyon, veri eklendikten sonra ağacın kök adresiyle geri dönmektedir.

```
BTREE* insertBST(BTREE *root, int data) {
    if(root != NULL){
        if(data < root -> data) // data, root'un datasından küçükse
            root -> left = insert(root -> left, data);
        else // data, root'un datasından büyük ya da eşitse
            root -> right = insert(root -> right, data);
    }
    else
        root = new_node(data);
    return root;
}
```

Bir Ağacın Düğümlerinin Sayısını Bulmak

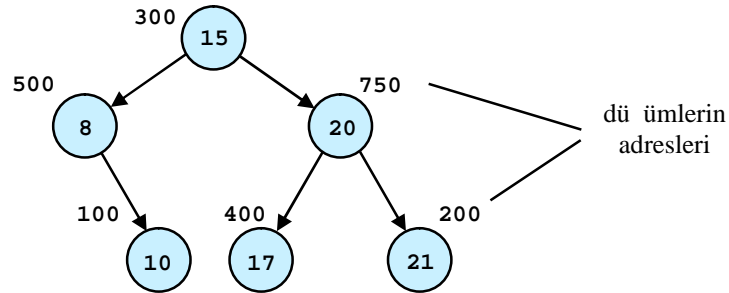
İNorder sıralama biçimine biraz benzemektedir. Sıralamada önce sol taraf yazdırılıyor, ardından kök ve sonra da sağ taraf yazdırılarak işlem tamamlanıyordu. Bu benzetimle ağaç üzerinde dolaşarak tüm düğümlerin sayısı bulunabilir. Fonksiyon rekürsif olarak modellendiğinde algoritma daha kolay bir ekil almaktadır. Geri dönüş değeri tamsayı olacaktır için `int` türden olmalı, input parametresi olarak ağacın kökünü almalıdır. Rekürsif fonksiyonlarda mutlaka bir çıkış yolu vardır ve çıkış yolu olarak `if-else` bloğunun kullanıldığını bahsetmiştik. İmdi bir ağacın düğümlerinin sayısını veren `size` isimli fonksiyonu yazalım.

```

/* İkili bir ağacın düğüm sayılarını veren fonksiyon */
int size(BTREE *root) {
    if(root == NULL)
        return 0;
    else
        return size(root -> left) + 1 + size(root -> right);
}

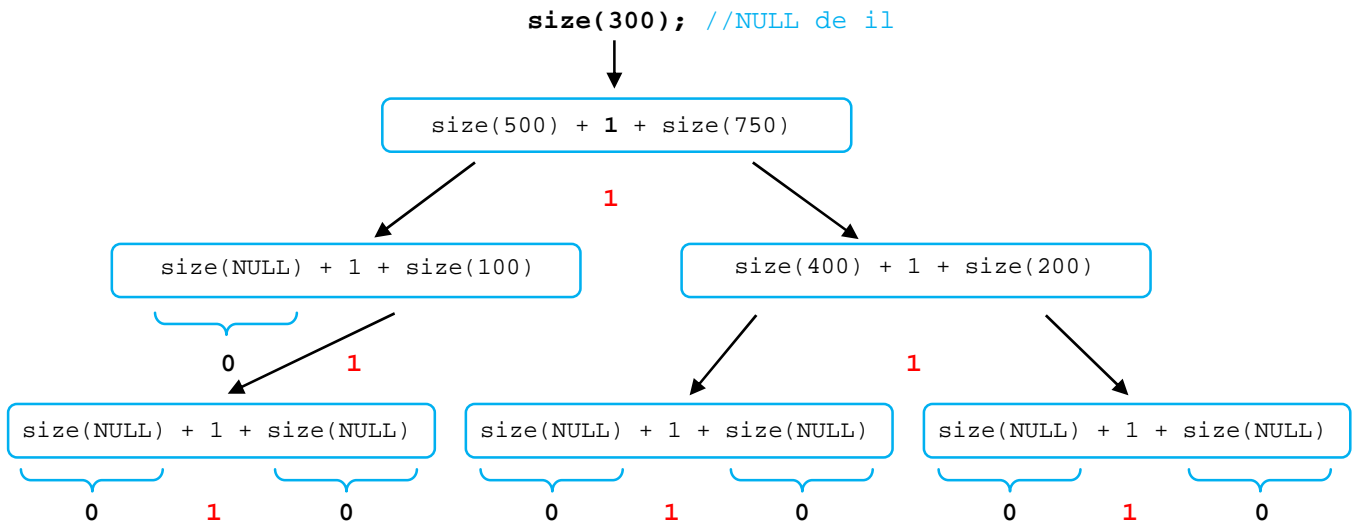
```

Fonksiyonun nasıl çalıştığını inceleyelim.



ekil 5.11 6 düğümü olan ikili bir arama ağacı.

Ağacımızda ekil 5.12'de de açıkça görüldüğü gibi `size(300)` çağrısıyla `root` `NULL` olmadığı için fonksiyonun `else` kısmı çalışıyor ve ilk olarak `size(root->left)` ile sol descendant dolaşıyor. Fonksiyondan geri dönen 0 ve 1 rakamlarının `stack`'te tutulduğu daha önceki konularda anlatılmıştı. Sonra da `size(root->right)` ile fonksiyonlar tekrar çağrılıyor ve sağ descendant dolaşıyor. **LIFO (Last in First Out) son giren ilk çıkar** prensibine göre `stack`'te tutulan rakamlar çıkarılacak ve toplanarak ağacın düğüm sayısı bulunacaktır.



ekil 5.12 `size()` fonksiyonunun çalışması.

Bir Ağacın Yüksekliğini Bulmak

Ağacın yüksekliği bir tamsayıdır, dolayısıyla fonksiyonun geri dönüş türü `int` olmalıdır. Parametre olarak yüksekliği bulunacak olan ağacın adresini almakta ve fonksiyon rekürsif olarak kendini çağırılmaktadır.

```

int height(BTREE *root) {
    if(root == NULL)
        return -1;
    else {
        int left_height, right_height;
        left_height = height(root -> left);
        right_height = height(root -> right);
        if(right_height > left_height)
            return right_height + 1;
        else
            return left_height + 1;
    }
}

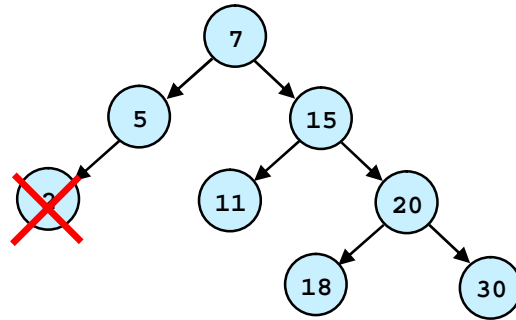
```

kili Arama A acından Bir Dü üm Silmek

Bir dü ümü silmek istedi imizde ilk olarak fonksiyona parametre olarak gönderdi imiz a açta eleman ya vardır veya yoktur. Eğer a açta hiç eleman yoksa fonksiyon NULL ile geri dönmelidir. A açta eleman var ise, silme i lemi ancak aranılan de ere bir e itlik varsa gerçekleşecektir. Yani bir a açtan 8'i silmek istiyorsak, ancak o a açta 8 içeren bir dü üm varsa silme i lemi gerçekleşir. Fakat silinecek dü üm a acın çocuklarından herhangi biri olabilir. Bu durumda silinecek dü üm, üzerinde bulunduğu dü ümden küçükse, sol tarafa doğru bir dü üm ilerleyip yeni bir kontrol yapmamız gerekir. Büyükse bu defa sağ tarafta bir dü üm ilerleyip kontrolü tekrarlamamız gerekmektedir. Aranılan dü üm bulunduğu da silme i leminden önce üç durumdan bahsetmeliyiz. kili bir arama a acında bir dü ümün en fazla iki çocu u olabilece ini dü ündü ümümüzde bunlar;

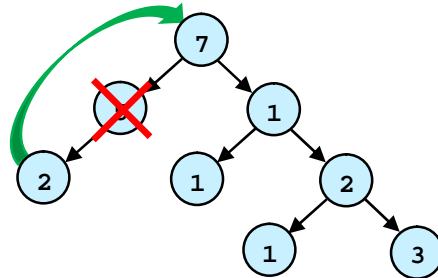
- 1- Silinecek olan dü ümün çocu u yok ise,
- 2- Silinecek olan dü ümün sadece bir çocu u var ise,
- 3- Silinecek olan dü ümün iki çocu u var ise,

nasıl bir algoritma kurmamız gerekti idir. Bir örnek ile gösterelim;



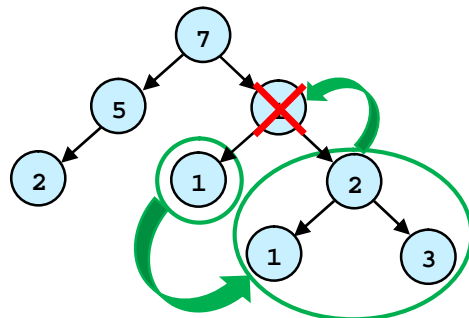
ekil 5.13 kili arama a acından bir yapra ın silinmesi.

Diyelim ki silinecek olan veriler yapraklardan (leaf) biri olsun. Yani 2, 11, 18 ya da 30'dan biri olsun. Bu durumda silinecek dü ümü direkt `free()` fonksiyonuyla yok edebiliriz. Yani 1 no'lu seçene in i oldukça kolay. Bir çocu u olan dü ümü nasıl silebiliriz? Örne in a a ıdaki ekilde görüldü ü gibi 5'i silmek isteyelim. Bu durumda dü ümün çocu unu ebeveynine ba lar ve istenen dü ümü silebiliriz. 2. seçene de kolay bir i lem gibi görünüyor.



ekil 5.14 kili arama a acından bir çocu u olan dü ümün silinmesi.

Peki, ekil 5.15'teki gibi 15'i silmek istersek ne olacak? Bir dü ümün kendisinin varsa sağ tarafındaki tüm çocuklardan küçük ya da e it oldu u bilindi ine göre, sağ çocuklarından kendisine en yakın de erdeki dü üm, sağ tarafındaki çocuklarının en küçük dü ümüdür. Öyleyse silinecek dü ümün sol çocukları, sağ tarafın en küçük çocu unun sol çocukları olacak ekilde ba lanır ve bu alt a aç silinecek dü ümün yerine kaydırılır.



ekil 5.15 kili arama a acından iki çocu u olan dü ümün silinmesi.

Bu algoritmaya göre silme i lemi yapacak fonksiyonu tasarlayabiliriz. Fonksiyon verilen dü ümü sildikten sonra a acı tekrar düzenleyip kökle geri dönece inden, geri dönü de eri root yani kök olmalı ve türü de BTREE olmalıdır. Parametre olarak hangi a açtan silme i lemi yapılacaksa o a acın kökünü ve silinecek veriyi (`x verisine sahip olan dü üm`) almalıdır. imdi `delete_node` isimli fonksiyonu yazalım.

```

BTREE *delete_node(BTREE *root, int x) {
    BTREE *p, *q;
    if(root == NULL)           A aç yoksa çalışacak olan kısım
        return NULL;
    if(root -> data == x) {
        if(root -> left == root -> right){
            free(root);
            return NULL;
        }
        else {
            if(root -> left == NULL) {
                p = root -> right;
                free(root);
                return p;
            }
            else if(root -> right == NULL){
                p = root -> left;
                free(root);
                return p;
            }
            else {
                p = q = root -> right;
                while(p -> left != NULL)
                    p = p -> left;
                p -> left = root -> left;
                free(root);
                return q;
            }
        }
    }
    else if(root -> data < x)
        root -> right = delete_node(root -> right, x);
    else
        root -> left = delete_node(root -> left, x);
    return root;
}

```

1.durum

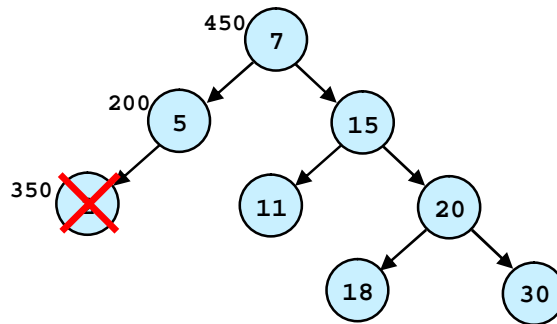
2.durum

3.durum

Aranan düüm bulunmuşsa çalışacak olan kısım

Aranan düüm henüz bulunamamışsa çalışacak olan kısım

Birinci durum için fonksiyonun rekürsif olarak çalışmasını anlatalım. Diyelim ki 2 verisine sahip düüm silinecek olsun.



ekil 5.16 kili arama a acında birinci durum görünümü.

ekilde düümün adresinin 350 olduğunu, kökün adresinin de 450 olduğunu görüyoruz. Kodu çalıştırdığımızda ilk olarak,

```
delete_node(450, 2);
```

1

çalışması yapılacak ve aranan düüm henüz bulunamadı için en altta bulunan else if blokları devreye girecektir. 7, 2'den küçük olmadığı için bir sonraki else kısmı yürütülecek ve orada da fonksiyon kendisini çağıracaktır.

```
else
```

```
root->left = delete_node(root->left, x); // Yürütülecek olan satır
```

450 adresinin left'i 200 adresini gösterdiğine göre,

```
delete_node(200, 2);
```

2

ça rısı yapılıyor. Dikkat edilirse fonksiyonun geri dönüşü de `root->left`'e atanacak. Aranana bulunamadı için en altta bulunan `else if` blokları tekrar devreye girecektir. 5, 2'den küçük olmadığı için bir sonraki `else` kısmı yürütülecek ve rekürsif olarak fonksiyon tekrar çalışacaktır.

```
else
    root->left = delete_node(root->left, x); // Yürütülecek olan satır
```

İmdi 200 adresinin `left`'i 350 adresini gösteriyor;

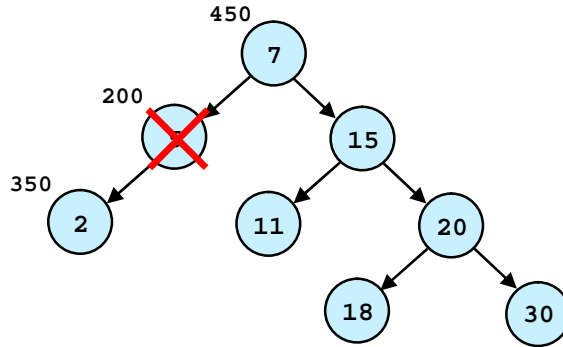
```
delete_node(350, 2);
```

3

Aranana bulunamadı şimdi bulunuyor. Dönüşüm bellekten siliniyor ve `root->left`'e `NULL` de atanıyor.

```
if(root -> data == x){ // root'un data'sı yani 2, 2'ye eşittir
    if(root -> left == root -> right) { // koşul doğru, ikisi de NULL
        free(root); // 350 adresine sahip dönüşüm bellekten silindi
        return NULL; // Fonksiyon NULL döndürerek sonlandı.
    }
    ...
    return root;
}
```

Fonksiyon bir önceki çağrıda noktaya (2 numaralı çağrı) geri dönüyor ve o noktadan sonra `return root` kodu çalışıyor. O sırada `root`'un adresi 200 olduğu için geriye 200 adresini döndürerek `root->left`'e atama yapıyor. Tekrar çağrıda bir önceki noktaya yani 1 numaralı bölüme dönerek sonraki kod olan `return root` icra edilerek, o esnadaki kök adresi olan 450 geri döndürüyor.



ekil 5.17 kili arama a acında ikinci durum görünümü.

ekil 5.17'de görülen ikinci durum için fonksiyonu tekrar çalıştırılm. 5 silinecek ve bu dönüşümün adresi 200, kökün adresi de 450'dir. İlk olarak kökün adresiyle fonksiyon çağrılıyor;

```
delete_node(450, 5);
```

1

Kökte 7 var, aranana bulunamadı için yine en altta bulunan `else if` blokları devreye giriyor. 7, 5'ten küçük olmadığı için bir sonraki `else` kısmı yürütülecek ve tekrar fonksiyon kendisini çağıracaktır.

```
else
    root->left = delete_node(root->left, x); // Yürütülecek olan satır
```

450 adresinin `left`'i 200 adresini gösterdiğine göre,

```
delete_node(200, 5);
```

2

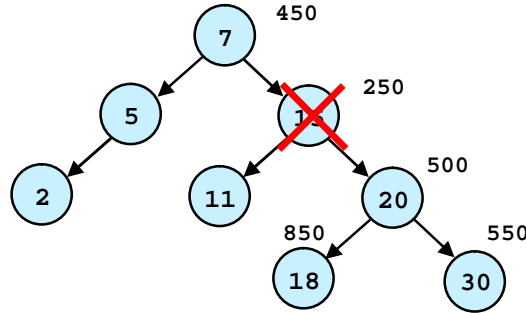
Aranana bulunamadı şimdi bulunuyor. Bu dönüşümün sol çocuğu var fakat sağ çocuğu yoktur. Bu yüzden aşağıda gösterilen `else` bloğu çalışıyor.

```
if(root -> left == NULL) {
    p = root -> right;
    free(root);
    return p;
}
else if(root -> right == NULL){
    p = root -> left; // Çalışacak blok
    free(root);
    return p;
}
```

Silinecek dönüşümün sağ çocuğu `NULL` olduğu için üstteki kodda görüldüğü gibi `else if` bloğu devreye giriyor. Sol çocuk `p` i adresine atanarak 5 siliniyor ve `p` i adresi geri döndürülüyor. Fonksiyon en son `delete_node(200, 5)`

çalışmasıyla 2 numara ile gösterilen bölümde çalışması ve return komutu icra edilerek p geri döndürülmüştü. Öyleyse ilk çalışması noktasına giderek p kökün sol çocuğu olarak atanıp henüz icra edilmemiş olan alttaki kodlar çalışacak ve kökün adresini geri döndürerek silme işlemi tamamlanmış olacaktır.

Son olarak ekil 5.18'de görüldüğü gibi üçüncü durum için fonksiyonu adım adım çalıştırıyoruz.



ekil 5.18 İkili arama ağacında üçüncü durum görünümü.

Silinecek olan veri 15 ve adresi 250, kökün adresi de 450'dir. İlk olarak kökün adresiyle fonksiyon çalışıyor;

```
delete_node(450, 15);
```

1

Kökte 7 var, aranan değer henüz bulunamadığı için yine en altta bulunan else if blokları devreye giriyor. 7, 15'ten küçük olduğu için fonksiyon bu defa ağacındaki sağ çocuğu ile kendisini çalıştıracaktır.

```
else if
```

```
    root->right = delete_node(250, 15); // Yürütülecek olan satır
```

2

Aranan değer bulundu. Yani if(root->data == x) kodundaki root'un data'sı da 15'tir, x de 15'tir. Sol ve sağ çocuğu NULL olmadığından dolayı fonksiyonda üçüncü durum olarak belirtilen else bloğu çalışıyor.

```
else {
    p = q = root -> right;
    while(p -> left != NULL)
        p = p -> left;
    p -> left = root -> left;
    free(root);
    return q;
}
```

Sol ve sağ çocuğun adresi p ile q işaretçisine atanıyor. Sonra silinecek değerün sağ soyundaki en küçük data'yı barındıran değerün adresi p işaretçisine atanıyor. Bu işlem while döngüsü içerisinde 15'in sağ çocuğu olan 20'nin sol tarafındaki yaprağa kadar inilerek sağlanıyor. İmdi p işaretçisinde bu en küçük veriyi barındıran değerün adresi, q işaretçisinde ise silinecek değerün sağ çocuğunun adresi tutuluyor (p=850, q=500). Daha sonra 15'in sol çocuğu olan 11'in adresi p->left = root->left atamasıyla, bulunan en küçük veriyi barındıran değerün sol çocuğu olarak başlanıyor. Ardından 15 siliniyor ve q işaretçisi geri döndürülüyor. Fonksiyon en son delete_node(250, 15) çalışmasıyla 2 numara ile gösterilen bölümde çalışması ve return komutu icra edilerek q geri döndürülmüştü. İmdi ilk çalışması noktasına giderek q kökün sağ çocuğu olarak atanıp henüz icra edilmemiş olan alttaki kodlar çalışacak ve o bölümdeki kökün adresi olan 450'yi geri döndürerek silme işlemi tamamlanmış olacaktır. Yeni oluşan ağaç ise hala bir **BST** (Binary Search Tree) ikili arama ağacıdır.

İkili Arama Ağacında Bir Değeri Bulmak

```
BTREE* searchtree(BTREE* tree, int data) {
    if(tree != NULL)
        if(tree -> data == data)
            return tree;
        else if(tree -> data > data)
            searchtree(tree -> left, data);
        else
            searchtree(tree -> right, data);
    else
        return NULL;
}
```

kili Arama A acı Kontrolü

```
boolean isBST(BTREE* root) { // boolean türü stack kısmında anlatılmıştı
    if(root == NULL)
        return true;
    if(root -> left != NULL && maxValue(root -> left) > root -> data)
        return false;
    if(root -> right != NULL && minValue(root -> right) <= root -> data)
        return false;
    if(!isBST(root -> left) || !isBST(root -> right))
        return false;
    return true;
}
```

kili Arama A acında Minimum Elemanı Bulmak

```
int minValue(BTREE* root) {
    if(root == NULL)
        return 0;
    while(root -> left != NULL)
        root = root -> left;
    return(root -> data);
}
```

kili Arama A acında Maximum Elemanı Bulmak

```
int maxValue(BTREE* root) {
    if(root == NULL)
        return 0;
    while(root -> right != NULL)
        root = root -> right;
    return(root -> data);
}
```

Verilen ki A acı Kar ıla tırmak

```
int isSame(BTREE* a, BTREE* b) {
    if(a == NULL && b == NULL)
        return 1; // İki a aç da boş ise true döndürür
    else if(a != NULL && b != NULL)
        return (
            a -> data == b -> data && isSame(a -> left, b -> left) &&
            isSame(a -> right, b -> right) // Koşul do ru ise true döndürür
        );
    else
        return 0;
}
```

Alı tırmalar

Örnek 5.5: Girilen bir x de eri, kökten itibaren yaprak dahil olmak üzere o yol üzerindeki verilerin toplamına e itse true, e it de ilse false döndüren path isimli programı kodlayınız.

```
int path(BTREE* root, int sum) {
    int pathSum;

    if(root == NULL) // A aç NULL ise
        return (sum == 0); // sum 0'a eşitse true dönüyor
    else {
        pathSum = sum - root -> data;
        return (
            path(root -> left, pathSum) ||
            path(root -> right, pathSum)
        );
    }
}
```

Örnek 5.6: Bir ikili arama ağacındaki verilerden tek olanları diğer bir BST ağacına kopyalayan `copyOdd` isimli programı yazınız.

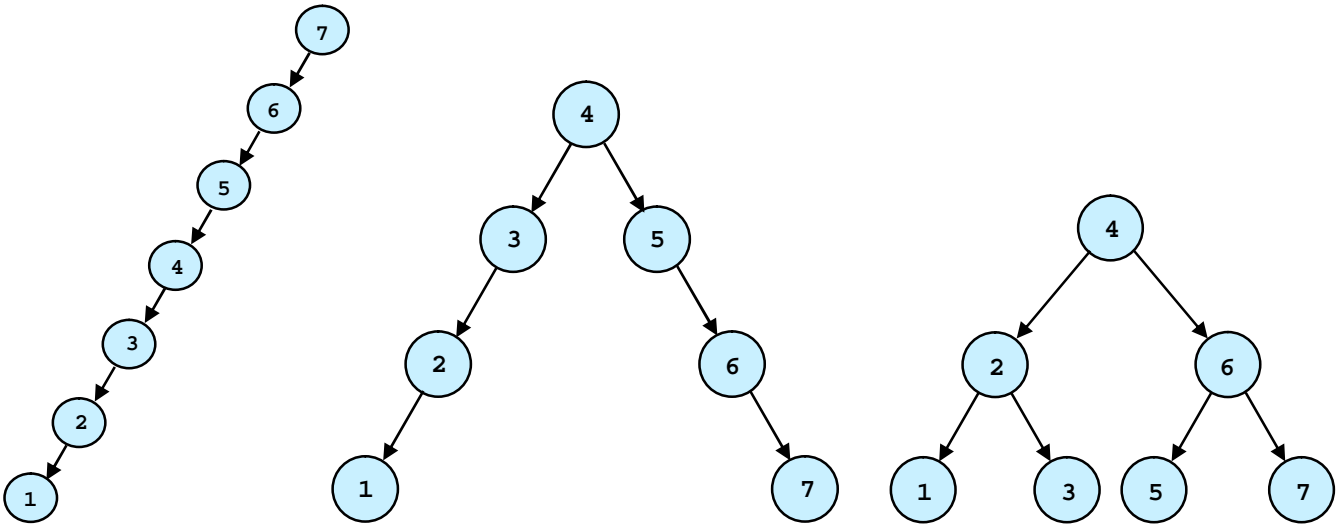
```
BTREE* copyOdd(BTREE* root, BTREE* root2) {
    if(root != NULL) {
        if(root->data % 2 == 1)
            root2 = insertBST(root2, root->data);

        root2 = copyOdd(root->left, root2);
        root2 = copyOdd(root->right, root2);
    }
    return root2;
}
```

5.4 AVL AĞAÇLARI

Yahudi bir matematikçi ve bilgisayar bilimcisi olan Sovyet Georgy Maximovich Adelson-Velsky (8 Ocak 1922 – 26 Nisan 2014) ile yine Sovyet matematikçi Yevgeny (Evgenii) Mikhaylovich Landis (6 Kasım 1921 – 12 Aralık 1997) adlı kişilerin 1962 yılında buldukları bir veri yapısıdır. Bu veri yapısı, isimlerinin baş harflerinden alıntı yapılarak AVL ağaçları olarak adlandırılmıştır. AVL ağaçları hem dengelidir hem de BST (*binary search tree*) ikili arama ağacıdır. Normal bir ikili ağacın yüksekliği maksimum n adet düğüm için $h = n - 1$ 'dir. AVL yöntemine göre kurulan bir ikili arama ağacında sağ alt ağaç ile sol alt ağaç arasındaki yükseklik farkı **en fazla bir** olabilir. Bu kural ağacın tüm düğümleri için geçerlidir. Herhangi bir düğümün sağ ve sol alt ağaçlarının yükseklik farkı 1'den büyükse o ikili arama ağacı, AVL ağacı değildir.

Normal ikili arama ağaçları için ekleme ve silme işlemleri ağacın orantısız büyümesine, yani ağacın dengesinin bozulmasına neden olabilir. Bir dalın fazla büyümesi ağacın dengesini bozar ve ağaç üzerine hesaplanacak karmaşık hesapları ve yürütme zamanı bantlarından sapılır. Dolayısıyla ağaç veri modelinin en önemli getirisi kaybolmaya başlar. Ağacımızda 1-7 arası sayıların farklı sırada ikili arama ağacına eklenmesiyle oluşan üç farklı ağaç gösterilmiştir.



ekil 5.19 a) kili bir arama ağacı.

ekil 5.19 b) kili baka bir arama ağacı.

ekil 5.19 c) Daha baka bir ikili arama ağacı.

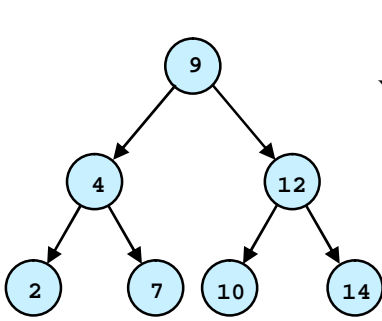
Sayıların hangi sırada geleceğini bilinemediğinden uygulamada bu üç ağacın biri ile karşılaşılabilir. Bu durumda dengeli ağaçlar tercih edilir. Dolayısıyla ağacın dengesi bozulduğunda ağacı yeniden dengelemek gerekir. kili arama ağacının yüksekliğinin olabildiğince düşük olması arzu edilir. Bu yüzden aynı zamanda **height balanced tree** olarak da bilinirler.

AVL ağacında bilinmesi gereken bir kavram denge faktörüdür (*balance factor*). Denge koşulu oldukça basittir ve ağacın derinliğinin $\theta(\lg n)$ kalmasını sağlar.

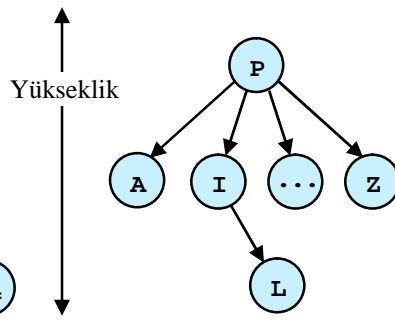
$$\text{Balance Factor} = \text{Height}_{(\text{right subtree})} - \text{Height}_{(\text{left subtree})}$$

Denge faktörü -1, 0 ve 1 değerini alabilir. Dikkat edilmesi gereken önemli bir nokta herhangi bir düğümün denge faktörü hesaplanırken sol ve sağ ağaçların yüksekliklerinin belirlenmesidir. Yoksa herhangi bir kayıt için düğümlerin sayılması değildir. Bir baka deyile AVL ağacında sol alt ağaç ile sağ alt ağaç farkının mutlak değeri 1'den küçük ya da 1'e eşit olmalıdır.

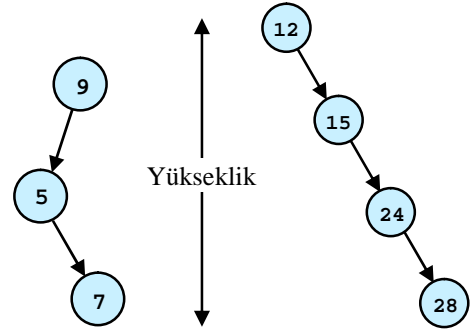
$$\lfloor \text{left} \rfloor - \lfloor \text{right} \rfloor = 1$$



ekil 5.20 a) Dengeli bir ikili ağaç.

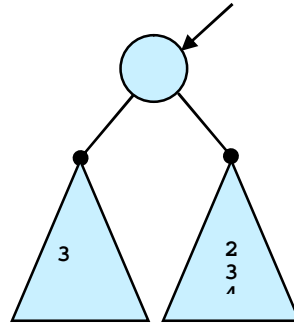


ekil 5.20 b) Dengeli bir ağaç.



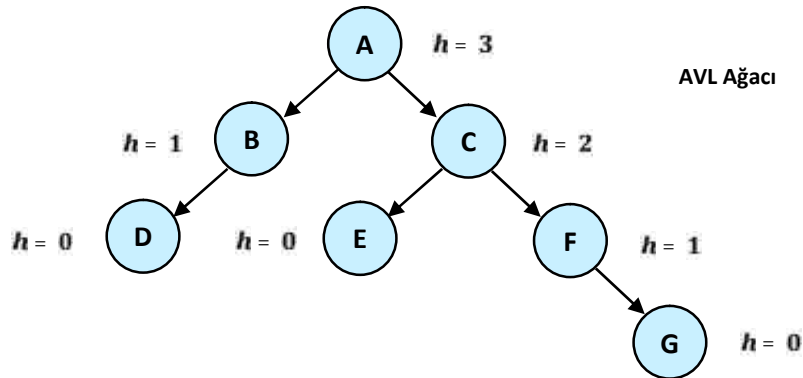
ekil 5.20 c) Dengesiz ikili ağaçlar.

Sağ veya sol çocuk yok ise (yani olmayan düğümlerin ya da diğer bir söyleyişle NULL değerlerin) yüksekliği -1 'dir. ekil 5.21'de sembolize edilen bir AVL ağacında üçgen biçimlerinde gösterilen sol alt ağacın yüksekliği 3 ise, sağ alt ağacın yüksekliği ancak 2, 3 veya 4 olabilir.



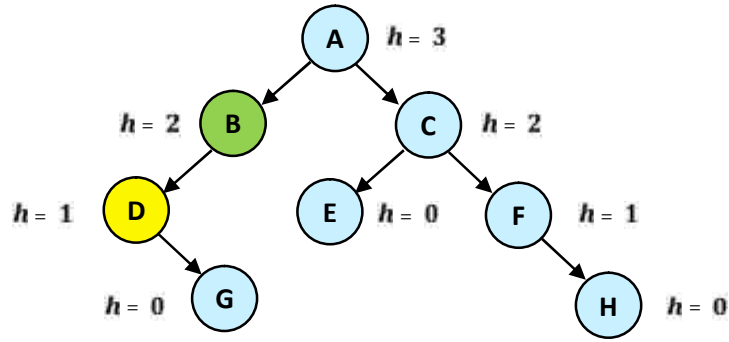
ekil 5.21 AVL ağacında sol ve sağ alt ağaçların yüksekliği.

ekil 5.22'de ise AVL ağacının sol ve sağ alt ağaçlarının her bir düğümünün yüksekliği kardeşiyle kıyaslanıyor. Kardeşi yoksa NULL olanın yükseklik değeri -1 olduğunu belirttik. Örneğin D düğümünün çocuğu yoktur. Dolayısıyla D düğümünün yükseklik değeri 0 'dır. Sağında düğüm yoktur ve yükseklik değeri -1 kabul edilmektedir. G yaprağının yüksekliği 0 'dır. Solunda düğüm yoktur ve yükseklik değeri -1 'dir. Diğer düğümlerin arasındaki yükseklik farkları aşağıdaki şekilde açıkça görülmektedir. Yüksekliği 1 olan düğümlere dikkat ediniz. Solundaki ya da sağındaki düğümle arasındaki yükseklik farkları 1 'dir.



ekil 5.22 AVL ağacında sol ve sağ alt ağaçların her bir düğümünün yüksekliği.

İimdi de AVL olmayan bir ağacı inceleyelim. ekil 5.23'te yer alan ikili ağaç dengeli bir ağaç değildir. Çünkü sarı renk ile belirtilmiş olan düğümün yüksekliği 1 iken, kardeşi olmadığından sağ tarafının yüksekliği -1 'dir ve arasındaki fark 2'dir. Bu nedenle bir AVL ağacı değildir. ekildeki yeşil renkli düğüm gibi tek çocuğu olan ve aynı zamanda torunu da olan ağaçlar bir AVL ağacı değildir de denilebilir.



ekil 5.23 Dengeli olmayan ikili bir ağaçta düğümlerin yükseklikleri.

Önerme:

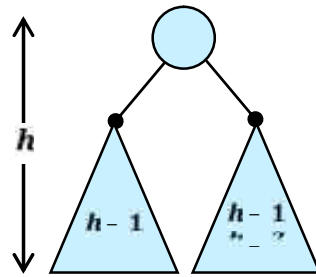
Bir ikili ağacın yüksekliği $\lg n < h < n$ iken, n düğümlü bir AVL ağacının yüksekliği $\theta(\lg n)$ 'dir. Burada θ , gayri resmi olarak merteye ya da civarı anlamındadır (θ gösteriminin formal tanımı, Algoritmalar dersinde verilecektir). Öyleyse ifadeyi, bir AVL ağacının yüksekliği 2 tabanında logaritma n civarındadır veya mertebesindedir şeklinde de söyleyebiliriz. $\lg n$, n 'den çok çok küçük bir sayıdır. Örneğin,

$n = 10^6$ ise $\lg 10^6 \approx 20$ civarındadır.

Bu önerme AVL ağaçları için bir kolaylık sağlamaktadır. Yüksekliğin az olması dengiyi sağlamak ve bu denge de arama, ekleme ve silme işlemlerinde yüksek bir performans kazandırmaktadır. Önermeyi ispatlamak için $f(h)$ fonksiyonunu tanımlayalım;

➤ $f(h)$, yüksekliği h olan bir AVL ağacındaki minimum düğüm sayısını ifade eder.

Kök ve iki çocuktan oluşan bir ağaçtaki düğüm sayısı 3'tür fakat tanımdaki minimum özelliğinden dolayı yüksekliği 1 olan AVL ağacının düğüm sayısı, minimum 2'dir. Örneğin $f(0)$ demek, AVL ağacının yüksekliği 0'dır demektir. Bu ağaçta sadece kök vardır. Düğüm sayısı 1'dir. $f(1)$ ise yani AVL ağacının yüksekliği 1 ise, bu ağaçta minimum 2 düğüm vardır. O halde $n = 2$ için, n yüksekliğindeki bir AVL ağacı, kök ve $n - 1$ yüksekliğinde bir alt ağaç ile yine $n - 1$ veya $n - 2$ yüksekliğinde diğer bir alt ağaç içerir.



ekil 5.24 AVL ağacında alt ağaçların yükseklikleri.

ekilde görülen AVL ağacında kökün sol çocuğunun yüksekliği $n - 1$ ise, sağ çocuğunun yüksekliği tanım gereği ancak $n - 1$ ya da $n - 2$ olabilir. Aradaki fark en fazla 1'dir. Sol alt ağacın yüksekliği $n - 1$ iken sağ alt ağacın yüksekliği n olamaz. Aradaki farkın 1 olması sizi yanıltmamalıdır. Çünkü ağacın yüksekliği n 'dir. Alt ağaçların n yüksekliğinde olması imkânsızdır.

spat: Bir AVL ağacının yapısı ekil 5.24'teki gibiyse bu ağaçta bir kök, sol alt ağaç ve sağ alt ağaç olacaktır;

$$f(n) = 1 + f(n-1) + f(n-2)$$

yazabiliriz. Sağ alt ağaç $f(n-1)$ veya $f(n-2)$ yüksekliğinde olabilir. Fakat tanımdaki minimum olma gereğinden dolayı $f(n-2)$ alınmalıdır. O halde $f(n-1) = f(n-2)$ 'dir. Eşitlik $f(n) = 1 + 2f(n-2)$ biçiminde de yazılabilir. Düzenlendiğinde;

$$f(n) > 2f(n-2)$$

olur. $f(n)$ 'teki h yerine n yazıldığında yer alan $n - 2$ yazıldığı gibi bu defa sağ taraf $2 \cdot 2f(n-2-2)$ 'den $4f(n-4)$ olur. Yine n yerine $n - 2$ koyarsak $2 \cdot 4f(n-2-4)$ 'ten $8f(n-6)$ olur. Bu işlemleri tekrarladığımızda ise baştan itibaren yeniden yazarsak;

$$\begin{aligned}
f(n) &> 2f(n-2) \\
f(n) &> 4f(n-4) \\
f(n) &> 8f(n-6) \\
f(n) &> 16f(n-8) \\
&\dots \\
&\dots
\end{aligned}$$

eklinde devam edecektir. Dikkat edilirse, e itsizli in sa ndaki fonksiyonun katsayıları 2'nin kuvvetleri eklinde devam etmektedir. Fakat kaç kadar devam edece i belli olmadığından katsayıyı 2^i eklinde ifade edebiliriz. Ayrıca h'tan çıkartılan sayıların 2'nin kuvveti olan i sayısı ile yine 2'nin çarpımları eklinde devam etmektedir. fadeyi i iterasyon kadar devam ettirirsek;

$$f(n) > 2^i f(n-2i) \text{ olur.}$$

Daha önce $f(0)$ 'ın 1'e e it oldu unu belirtmi tik. İmdi e itsizli in sa tarafından h - 2i, 0'a e itlendi inde, h - 2i = 0 e itli inden i = h/2 bulunur. Bulunan n/2'yi üstteki ifadede yer alan i'nin yerine yazıldı nda ise;

$$f(n) > 2^{h/2} f(0) \quad f(n) > 2^{h/2}$$

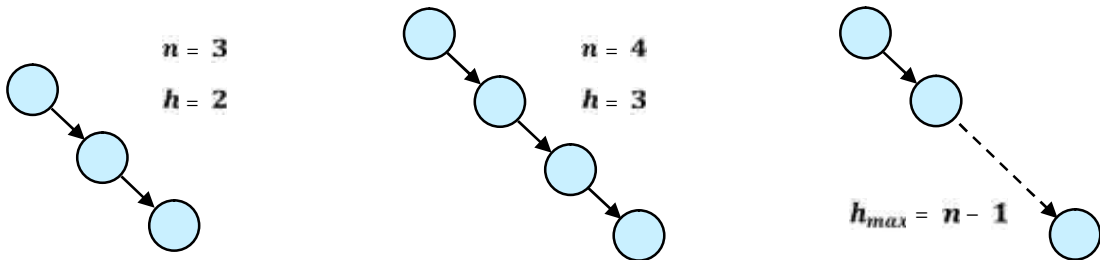
elde edilir. Her iki tarafın 2 tabanında logaritması alınırsa;

$$\lg f(n) > \frac{n}{2}$$

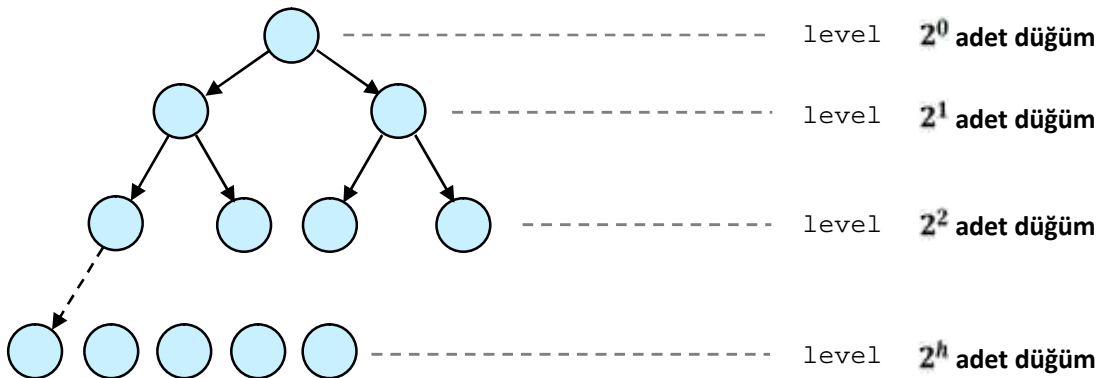
olur. Buradan da $n < 2 \lg f(n)$ yazılabilir. n dü ümlü bir ikili a acın yüksekli i en az $\lg n$ olaca ndan, AVL a acının yüksekli i $\theta(\lg n)$ 'dir.

Örnek 5.7: n dü ümlü bir ikili a acın maksimum ve minimum yüksekli ini bulunuz.

Çözüm: kili a acın maksimum yüksekli ini bulmak oldukça kolaydır. Bir ikili a acın maksimum yüksekli e sahip olabilmesi için a a daki ekilde görüldü ü gibi dü ümlerin ard ına sıralanması gerekir.



ekilde 3 dü ümlü bir ikili a acın yüksekli i 2, 4 dü ümlü bir ikili a acın yüksekli i ise 3'tür. O halde n dü ümlü bir a acın maksimum yüksekli i $h_{\max} = n - 1$ 'dir. Bir ikili a açta minimum yüksekli i sa lamak için bir seviye tamamlanmadan di er seviyeye dü üm ba lanmamalıdır. Mümkün oldu u kadar a acı sıkı ik yapmak gerekmektedir. Böyle bir ikili a açta bir dü ümün kök dü ümden olan uzaklı ma düzey (level) dendi i bilindi ine göre kök dü ümün düzeyi 0 (sıfır), yaprakların düzeyi ise n'a e it olacaktır.



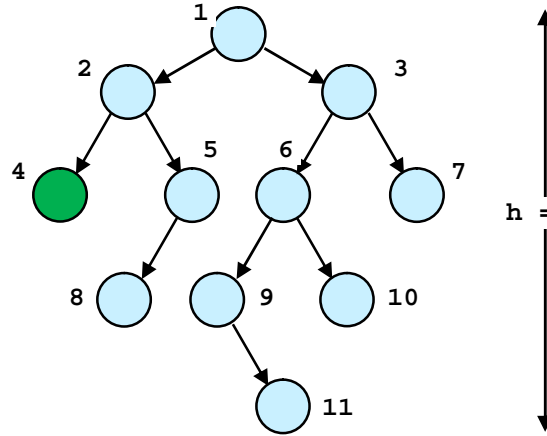
Son düzeydeki dü üm sayısının en fazla 2^h adet olabilece ine dikkat ediniz. Öyleyse bu ekilde bir h yüksekli indeki ikili a açta olabilecek maksimum dü üm sayısı, $1 + 2 + 2^2 + \dots + 2^h$ geometrik serisi olacaktır. Bu da $2^{h+1} - 1$ 'e e ittir. kili bir arama a acındaki n adet dü ümle n yükseklik ili kisini $2^{h+1} - 1 \approx n$ eklinde ifade edebiliriz. Düzenlendi inde ise, $2^{h+1} \approx n + 1$ eklini alan ifadenin her iki tarafının 2 tabanında logaritmasını aldı mızda;

$$\lfloor \frac{n}{2} \rfloor + 1 \quad \lg n + 1 \quad \lfloor \frac{n}{2} \rfloor \quad \lg n + 1 - 1$$

olur. Buradan ikili bir ağacın minimum yüksekliği $\lfloor \lg n \rfloor + 1$ bulunur. Bulunan bu sonuç, ikili bir ağacın maksimum yüksekliğinin n mertebesinde, minimum yüksekliğinin ise $\lg n$ mertebesinde olduğunu göstermektedir. Yükseklik n ile $\lg n$ arasında da ilişki vardır.

Bir AVL Ağacının Yapısı

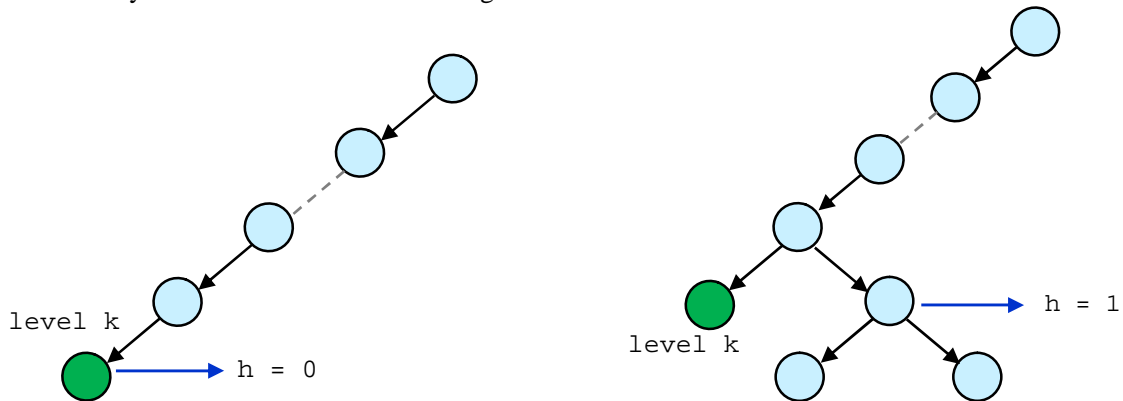
n düğümlü bir AVL (denge bir BST) ağacının yüksekliği h olsun. Köke en yakın yaprak k seviyesinde olsun. Köke en yakın yaprak k seviyesindeyken bir ağacın yüksekliği $k+1, k+2, k+3, \dots$ değerlerini alabilirken en fazla $2k$ değerinde olabilir. Şekil 5.25'te görülen AVL ağacında köke en yakın yaprak 4 numaralı düğüm olsun. Bu düğüm level 2 seviyesindedir. Bazı kaynaklarda seviyelerin 1'den başladığı söylenir. Fakat bunu söyleyenlerin oranı oldukça düşüktür. Biz kendi kabulümüzde ilk seviyeyi, yani kökün düzeyini 0 olarak alacağız. İmdi bu düğümün en yakın yaprak olduğunu varsayalım. En yakın yaprak demek, diğer yapraklar daha üst seviyede değil, daha üstte daha yakın yapraklar yoktur demektir. Peki, bu ağacın yüksekliği kaç olabilir? Tabii ki en az 2 olmalıdır. Evet ama en fazla ne kadar olabilir? Bir AVL ağacını çizerek konuyu daha iyi kavrayalım.



Şekil 5.25 AVL ağacının $2k$ yüksekliğinde olması.

Şekildeki AVL ağacında köke en yakın yaprak 4 numaralı yeşil olan yapraktır. Yüksekliği 0'dır. Kardeşi düğümün yüksekliği ise 1 olup aralarındaki fark 1'dir. Şekilde de görüldüğü gibi ağaçta toplamda 11 düğüm bulunmaktadır. Ağacın AVL özelliği bozulmadan ne kadar düğüm eklenebilir? Şekilde 11 numaralı düğüme herhangi bir düğüm eklenemez. Eğer eklenirse AVL özelliği kaybolacaktır. Böyle bir durumda 9 numaralı düğümün yüksekliği 2, 8 ve 10 numaralı düğümlerin yüksekliği ise 0 olacaktır için denge kavramı ortadan kalkacaktır. 11'e düğüm eklenirse 8 ve 10 numaralı düğümlere de ekleme yapılmalıdır. 8 ve 10'a ekleme yapıldığında bu defa da 4 ve 7 numaralı düğümlerle olan yükseklik derinleşecek, onlara da ekleme yapılırsa köke en yakın yaprak da derinleşecektir. Bir AVL ağacının yüksekliğinin en fazla $2k$ olduğunu ispatlamak için aşağıda görülmüştür.

spat: Yeşil düğümü k seviyesinde ve amaç, ağacın maksimum yüksekliğe ulaşmasını sağlamaktır. Ağacın üst seviyelerindeki düğümler ve dallar gösterilmemiştir.

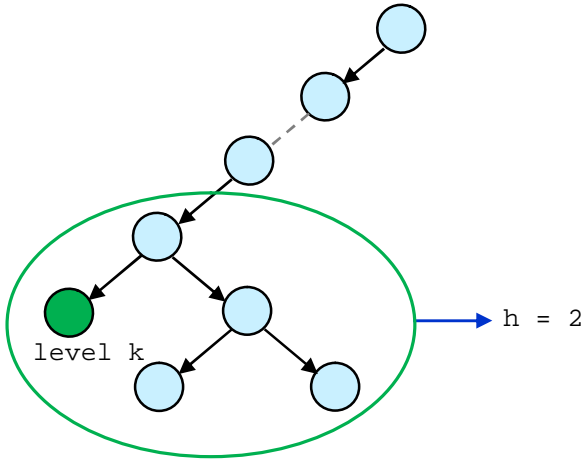


Şekil 5.26 a) AVL ağacında k seviyesindeki yaprak.

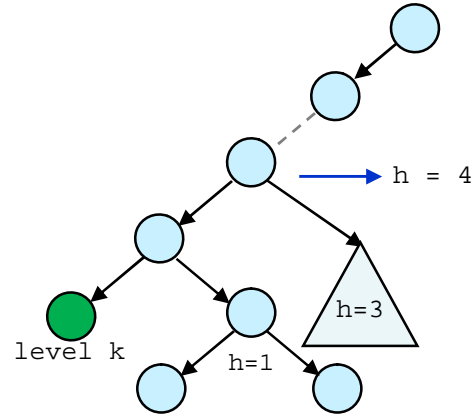
Şekil 5.26 b) k seviyesinde bir yaprak varken ekleme yapmak.

Şekil 5.26 a)'da ki AVL ağacına dengeyi bozmayacak şekilde en fazla Şekil 5.26 b)'de ki gibi ekleme yapılabilir. Daha fazla düğüm eklendiğinde ya denge bozulacak ya da köke en yakın yaprak derinleşecektir. Şekil 5.27 a)'da bir AVL ağacında sol alt ağaç daire içine alınmıştır. Şekil 5.27 b)'de ise sağ taraftaki üçgenle gösterilen alt ağaç temsil edilmiştir. Eğer sol alt

a aç 2 yüksekli indeyse denge gere i sa alt a acın yüksekli i **en fazla** 3 olabilir. Sa alt a acın ba lı oldu u dü ümün yüksekli i ise 4'tür.

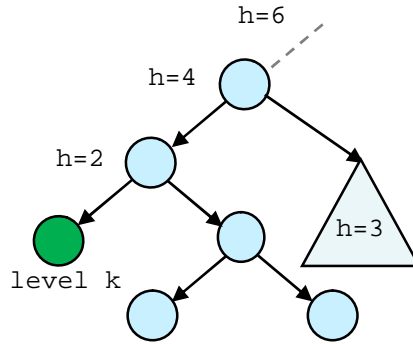


ekil 5.27 a) Daire içine alınmış alt a acın yüksekli i 2'dir.



ekil 5.27 b) Üçgenle gösterilen alt a acın yüksekli i 3'tür.

Bir üst seviyeye çıktığımızda ise sa alt a acın yüksekli i **en fazla** 5 olabilirken ba lı oldu u ebeveyninin yüksekli i 6'dır ve bu eklede devam eder.



ekil 5.28 Köke en yakın yaprak ve atalarının yükseklikleri.

Kökün yüksekli i bilinmedi i için x 'le gösterilirse a a ıdaki tablo ortaya çıkacaktır. ekil 5.28'deki örnekten devam edersek, köke en yakın yaprak k seviyesindeyken yüksekli i 0'dır. Yaprak ın 2 yüksekli indeki ebeveyninin seviyesi ise $k-1$ 'dir. Dikkat edilirse yaprak ın her atasının yükseklikleri 2'er artmaktayken, seviye ise 1 azalmaktadır. Toplamda 0.seviye ile birlikte $k+1$ kadar sayı vardır. Öyleyse yükseklikteki x 'in de eri $2k$ olur.

Tablo 5.1 Seviyedeki her bir azalmaya karşı yükseklik 2 kat artıyor.

Yükseklik	0	2	4	6	8	x
Seviye	k	$k-1$	$k-2$	$k-3$	$k-4$	0

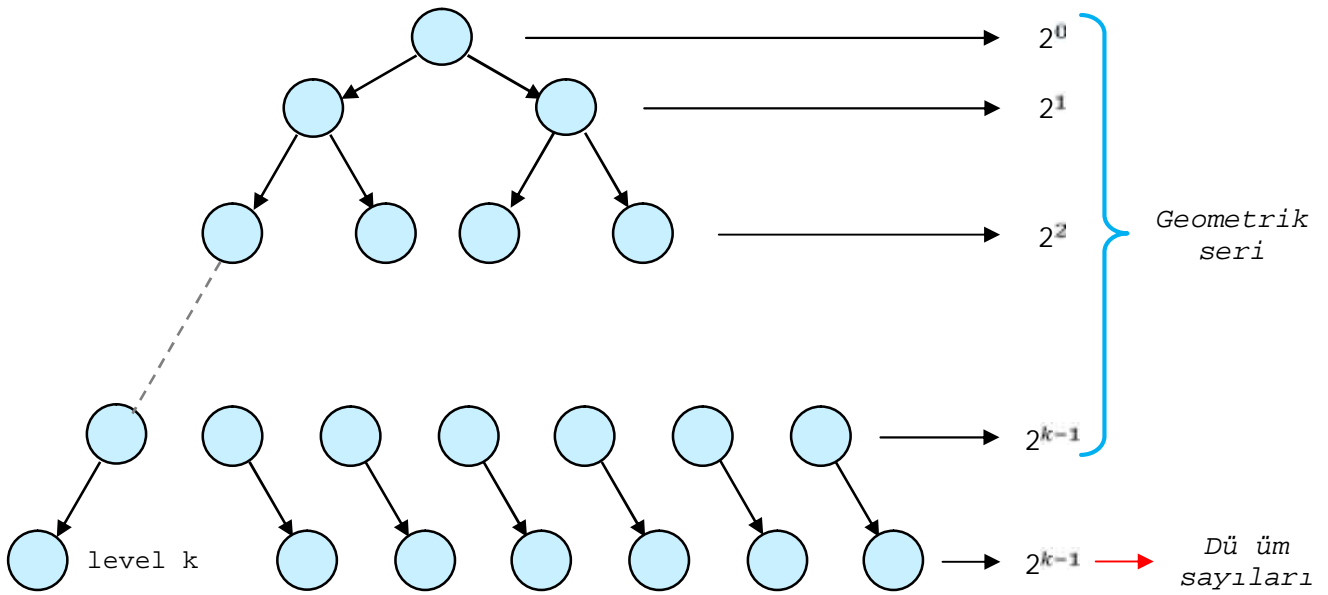
ddia: Köke en yakın yaprak k seviyesinde ise 0, 1, 2, ..., $k-2$ seviyelerindeki dü ümlerin 2 çocu u vardır.

spat: Bu iddiayı ispatlamak için çeli ki yöntemini kullanabiliriz. $k-2$ seviyesindeki u dü ümünün sadece bir v çocu u oldu u kabul edilsin. O halde v , $k-1$ seviyesindedir ve yaprak olamaz. Bu yüzden v dü ümünün **en az** bir çocu u vardır. Bir çocu unun olması ise u dü ümünde bir yükseklik ihlali demektir ve çeli ki elde edilir.



Şekil 5.29 $k-2$ seviyelerindeki dü ümlerin 2 çocu u vardır.

Bu iddia $0, 1, 2, \dots, k-1$ seviyelerinin tamamen dolu olduğunu göstermektedir. Köke en yakın yaprak k seviyesindeyken a açtaki **minimum** eleman sayısı da hesaplanabilir. A a ıdaki şekilde hesaplama mantığı gösterilmiştir.



ekil 5.30 Köke en yakın yaprak k seviyesindeyken a açtaki minimum düüm sayısı.

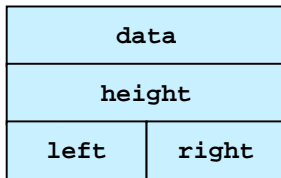
ekil 5.30'da görüldü ü gibi a açtaki düüm sayısının **minimum** olabilmesi için, köke en yakın yaprak düzeyinden daha a a ısında düüm olmaması gerekir. k düzeyinde ise yine **minimum** olma özelli inden dolayı birer yaprak bulunmalıdır ve bu da a acın k ve $k-1$ düzeylerinde aynı sayıda düüm olması demektir. $2^0, 2^1, 2^2, \dots, 2^{k-1}$ toplamları geometrik bir seridir ve bu toplam $2^k - 1$ sayısına e ittir. A açtaki minimum düüm sayısı ise bu geometrik seri ile k düzeyinin düüm sayısının toplamına e ittir.

$$\text{minimum düüm sayısı} = 2^k + 2^{k-1} - 1$$

Artık bir AVL a acının genel yapısını tanımlayabiliriz.

```
struct node {
    int data;
    int height;
    struct node *left;
    struct node *right;
};
typedef struct node AVLTREE;
```

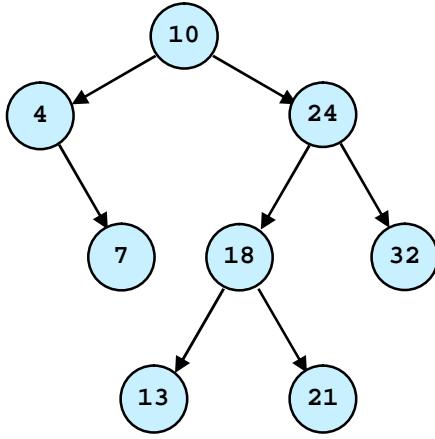
typedef bildirimini yaparak struct node yerine bundan sonra AVLTREE kullanacağız.



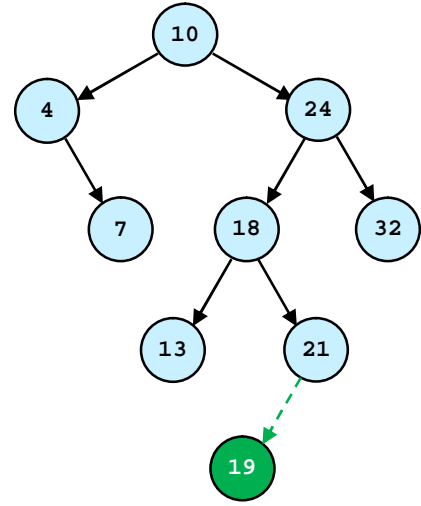
ekil 5.31 Bir AVL düümünün mantıksal gösterimi.

AVL A açlarında Ekleme lemi

AVL a açlarına ekleme yapılırken hem denge hem de BST özelli i kaybolmamalıdır. Ekleme i leminde bazı düümlerin yükseklikleri de i e bilece inden a acı tekrar denge durumuna ayarlamak gerekir. A a ıda ekil 5.32 a)'da BST tarzında bir AVL a acı görülüyor. Aynı a aca ekil 5.32 b)'de görüldü ü üzere 19 içeri inin eklenmesi halinde a açta denge kavramı ortadan kalkıyor. 24 içeri inin bulundu u düümün yüksekli i ile 4 içeri inin bulundu u düümün yüksekli i arasındaki fark 2'ye çıkıyor. Daha önce anlatmı oldu umuz ikili arama a açlarına düüm ekleme i lemindeki gibi rekürsif bir fonksiyonla AVL a acını denge durumuna sokmaya çalı mak oldukça maliyetlidir. Örne in bir a açta 1 milyon adet veri bulunabilir. Bu kadar veriyi düzenlemeye çalı mak çok zahmetli olacaktır.

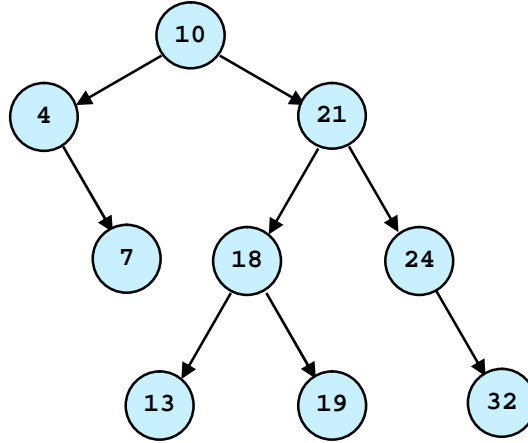


ekil 5.32 a) Henüz dengede olan bir AVL ağacı.



ekil 5.32 b) 19 içeriğinin eklenmesiyle denge bozuluyor.

Sorunu çözmek için ağaçta bir döndürme (*rotasyon*) uygulanmalıdır. ekil 5.33'te bir dizi döndürme uygulanarak AVL ağacındaki problemin giderildiği görülmüyor.



ekil 5.33 19 eklendikten sonra döndürme uygulanmış AVL ağacı.

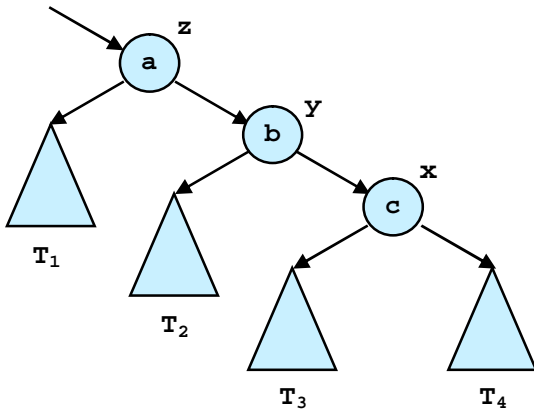
AVL ağaçlarında ekleme işleminde ağacı döndürmek gerektiğinden ekleme fonksiyonunu döndürme konusunu anlattıktan sonra yazacağız.

Bir AVL Ağacında Döndürme İşlemleri Döndürmek

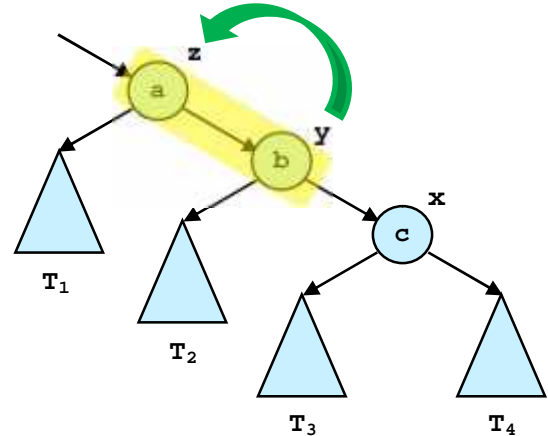
AVL ağaçlarında döndürme işlemi için ana olarak iki tane döndürme işlemi olmakla birlikte 4 farklı yol bulunmaktadır. Birisi **single** (*tek*) rotation ve diğeri ikisi de **double** (*çift*) rotation işlemidir. Her iki döndürme biçimi aslında aynı olmakla birlikte birbirinin simetriğidir ve biri sol taraftan, diğeri sağ taraftan döndürme işlemi yapmaktadır. Double rotation aynı zamanda single rotation'ı da içermektedir.

Tek Döndürme (Single Rotation)

ekil 5.34 a)'da görüldüğü gibi alt ağaçları olan bir AVL ağacında T_3 alt ağacına bir düğüm eklendikten sonra meydana gelen AVL ağacı kural ihlalinin **a** düğümünde olduğunu varsayalım. İhlalin meydana geldiği düğüme **z**, düğümün torununa da **x** diyeceğiz. **c** düğüme **x** ve arada kalan düğüme de **y** adını vereceğiz. Hemen ağla **a**'nın birden çok torunu olduğunu ve neden **c** düğüme **x** adının verildiğini ağla gelebilir. Nedeni basittir, çünkü düğüm ekleme işlemi T_3 tarafına yapılmıştır ve ihlal olduğu yol üzerinde bulunan torun **c** düğümdür. **x**, **y** ve **z** olarak isimlendirilen düğümler **zig-zig** durumundadır. Böyle bir durumda **z** düğüme sola döndürme (*left rotate*) işlemi uygulanır. Sola döndürmenin nasıl yapılacağı ekil 5.34 b)'de görülmektedir. **b** düğümünden itibaren saat yönünün tersine yani sola doğru 90° döndürülür.



ekil 5.34 a) Denge durumunda bir AVL alt ağacı.

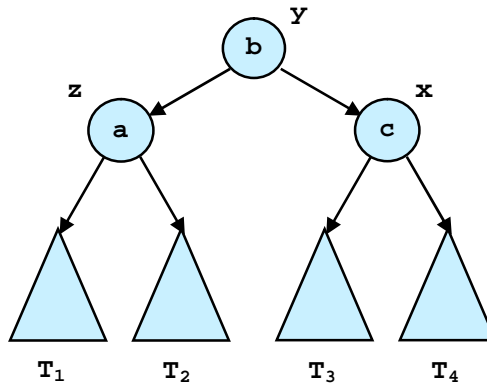


ekil 5.34 b) Ekleme yapıldığında zig-zig yapısında ki AVL alt ağacı.

Döndürme sonucunda b düğümü alt ağacın ebeveyni (*parent*) durumuna, a düğümü ve torunları ise b düğümünün alt soyu (*descendant*) haline gelmektedir. Alt ağaçların son durumu ekil 5.35'te görülmektedir. Döndürme işleminden sonra diğer alt ağaçların yerleştirilmesi BST'ye uygun biçimde yapılmalıdır. Yukarıdaki ekleme göre alt ağaçlar;

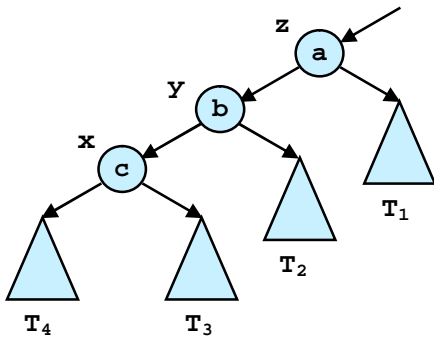
$T_1 \quad z \quad T_2 \quad y \quad T_3 \quad x \quad T_4$

ekilde sıralanırlar. Burada alt ağaçlar sıralı bir biçimde dizilmiş fakat her zaman bu şekilde sıralı olmayabilir.

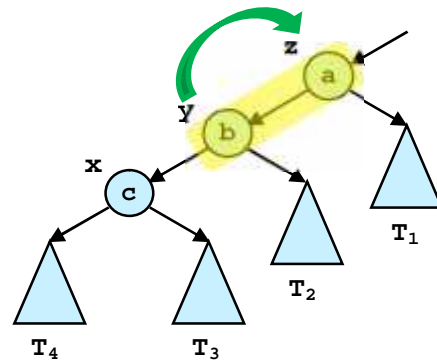


ekil 5.35 Sola döndürülmüş AVL alt ağacı.

İimdi de tek döndürmenin (*single rotation*) simetrisi olan sağa döndürme (*right rotate*) işlemini inceleyelim. ekil 5.36 a)'da yer alan T_3 ya da T_4 alt ağaçlarından birine ekleme yapılmak istenirse, hâlin görüldüğü düğümü yine z olarak isimlendireceğiz ve hâlin olduğu düğümünden ekleme yapılan düğüm üzerindeki yol üzerinde bulunan toruna da x ismini vereceğiz. Arada kalan çocuk düğüm ise yine y adını alacaktır.



ekil 5.36 a) Denge durumunda bir AVL alt ağacı.

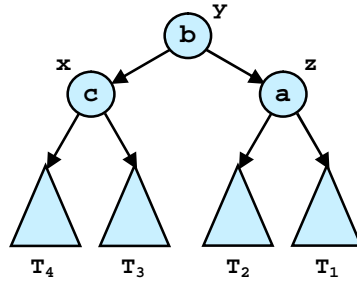


ekil 5.36 b) Döndürme uygulanacak zig-zig yapısında AVL alt ağacı.

Bu durumda da z düğümüne sağa döndürme (*right rotate*) işlemi uygulanır. Sağa döndürmenin nasıl yapılacağı ekil 5.36 b)'de görülmektedir. b düğümünden itibaren saat yönünde yani sağa doğru 90° döndürülür. Döndürme sonucunda b düğümü alt ağacın ebeveyni (*parent*) durumuna, a düğümü ve torunları ise b düğümünün sağ alt soyu (*descendant*) haline gelmektedir. Döndürme işleminden sonra diğer alt ağaçların yerleştirilmesinin BST'ye uygun biçimde yapılması unutulmamalıdır. Sıralamaları ise;

$$T_4 \quad x \quad T_3 \quad y \quad T_2 \quad z \quad T_1$$

eklinedir. Alt ağaçların son durumu ekil 5.37'de görülüyor.

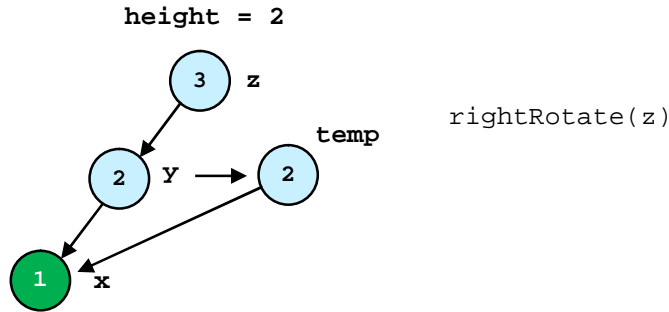


ekil 5.37 Sağa döndürülmüş AVL alt ağacı.

Single right rotation için `rightRotate` isimli fonksiyonun kodları aşağıda görüldüğü gibi yazılabilir;

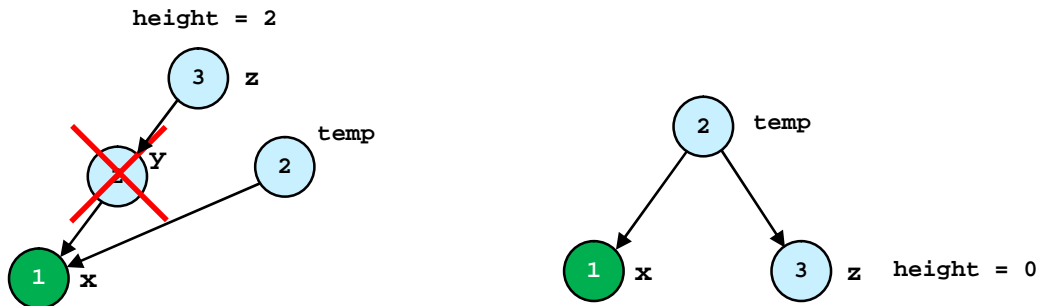
```
AVLTREE *rightRotate(AVLTREE *z) {
    AVLTREE* temp = z -> left; // z düğümünün sol çocuğu temp'e atanıyor
    z -> left = temp -> right; // temp'in sağ çocuğu z'nin sol çocuğu olarak atanıyor
    temp -> right = z; // son olarak z düğümü temp'in sağ çocuğu olarak atanıyor
    // Yükseklikler güncelleniyor
    z -> height = maxValue(height(z -> left), height(z -> right)) + 1;
    temp -> height = maxValue(height(temp -> left), height(temp -> right)) + 1;
    return temp;
}
```

Fonksiyonun geri dönüş değeri `AVLTREE*` olduğu için türü de `AVLTREE*`'dir. Parametre olarak `z` olarak işaretlediğimiz imiz döndürülecek düğümün adresini almaktadır. ekil 5.38'de görüldüğü üzere 3 ve 2 değerlerinin bulunduğu AVL ağacına 1 eklendiğinde ağacın dengesi bozulacağından sağa döndürme işlemi yapılmalıdır. Fonksiyonda ilk olarak `temp` isimli `AVLTREE*` türünden bir değeri kenara tutuluyor ve `y` adını verdiğimiz `z` düğümünün sol çocuğu bu değeri kene atanıyor. `temp` değeri hem `z` düğümünün, hem de `temp`'in sol çocuğu olarak yeni eklenen 1 düğümüdür.



ekil 5.38 z düğümünün ilk baştaki yüksekliği 2'dir.

Daha sonra `z`'nin sol çocuğu olan `y` düğümüne `temp`'in `right` tarafı atanıyor. ekil 5.39 a) ve b)'de de görüldüğü gibi `temp`'in sağ çocuğu olmadığı için `y` düğümüne `NULL` değeri atanması olacaktır.



ekil 5.39 a) z ile y düğümü arasındaki bağlantı koparılıyor.

ekil 5.39 b) x ve z artık temp'in çocukları durumundadır.

Nihayetinde `z` düğümü `temp`'in sağ tarafına bağlanıyor ve `z` düğümünün yüksekliği 0 ($2 - 2 = 0$) olarak güncelleniyor. Single left rotation için `leftRotate` isimli fonksiyonu da benzer biçimde yazabiliriz.

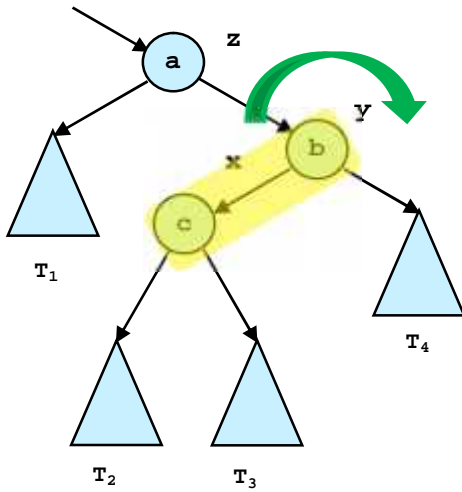
```

AVLTREE *leftRotate(AVLTREE *z) {
    AVLTreeNode *temp = z -> right; // z düümünün sa çocu u temp'e atanıyor
    z -> right = temp -> left; // temp'in sol çocu u z'nin sa çocu u olarak atanıyor
    temp -> left = z; // son olarak z düümü temp'in sol çocu u olarak atanıyor
    z -> height = maxValue(height(z -> left), height(z -> right)) + 1;
    temp -> height = maxValue(height(temp -> left), height(temp -> right)) + 1;
    return temp;
}

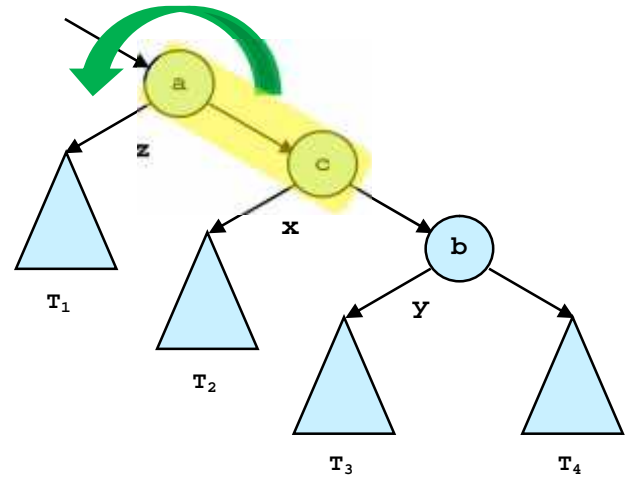
```

Çift Döndürme (Double Rotation)

Single rotation ilemindeki gibi çift döndürme ileminde de ekleme yapıldı ı zaman olu an ihlalin görüldü ü ilk dü üme z adı verilir. hlahin görüldü ü dü ümden ekleme yapılan dü üme do ru olan yol üzerinde bulunan toruna da bildi iniz gibi x ismi ve arada kalan çocuk dü üme de y ismi verilmektedir. ekil 5.40 a)'da **zig-zag** tipinde bir alt a aç görülmektedir. Bu alt a açta T_2 ya da T_3 'den birine ekleme yapıldı ında AVL özelli i ihlalinin görüldü ü a dü ümü z olarak, di er dü ümler de y ve x olarak adlandırılmı ır. Double rotate mantı mı daha iyi anlamak için iki kural iyi bilinmelidir. İki, a aç b dü ümünden itibaren BST özelli i de dikkate alınarak bir kez sa a döndürülmeli ve **zig-zig** durumuna getirilmelidir. kinci olarak ise a aç sa a döndürüldükten sonra z olarak adlandırılan a dü ümünden itibaren sola döndürülmelidir.

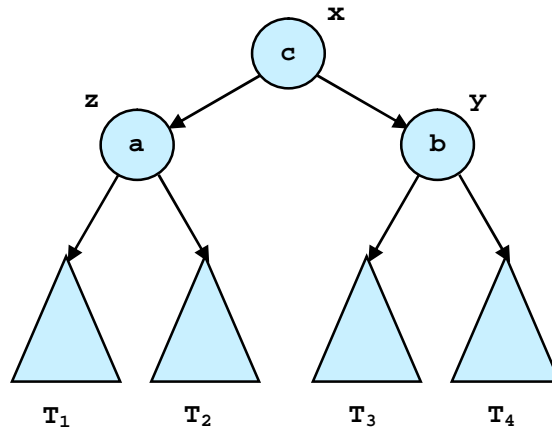


ekil 5.40 a) A aç sa a döndürülmelidir.



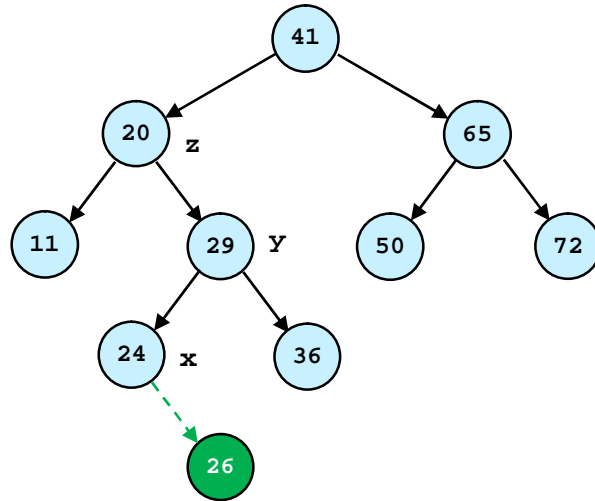
ekil 5.40 b) A aç imdi sola döndürülmelidir.

ekil 5.40 a)'da a acın ilk durumu, ekil 5.40 b)'de ise bir kez sa a döndürülerek **-right rotate R-R(b)**- zig-zig biçimine getirilmi a acı görüyorsunuz. A aç bu kez sola döndürülerek **-left rotate L-R(a)**- double rotation ilemi tamamlanmı olacaktır. A acın en son durumu ekil 5.41'de görülmüyor.



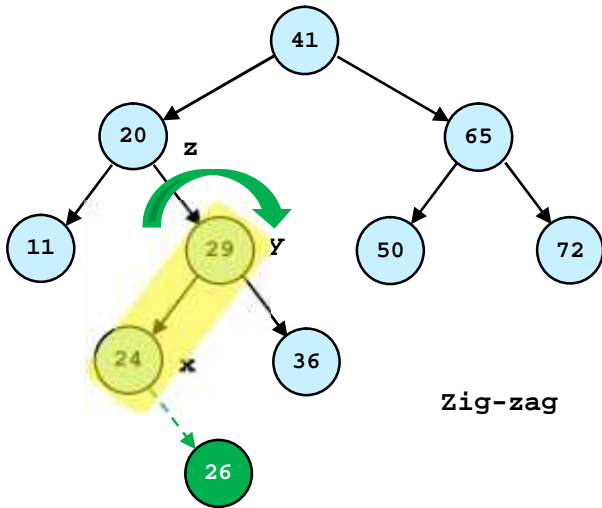
ekil 5.41 Double rotation uygulanmı alt a aç.

Double rotation'ın simetri i olan döndürme yönteminde de farklı bir i lem yoktur. Bir örnekle çift döndürme ileminin simetri ini de inceleyelim. Örnek a acımız ekil 5.42'deki gibi olsun.

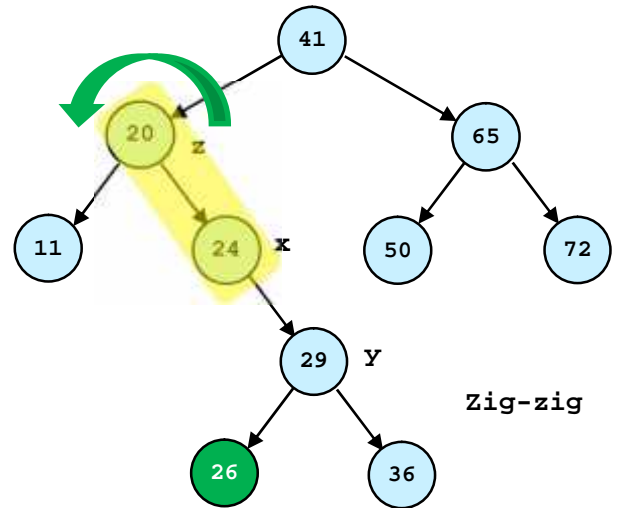


ekil 5.42 Alt a ca 26 eklenmek isteniyor.

ekilde görülen a ca 26 sayısı eklenmek istendi inde ilk olarak BST'ye uygun bir biçimde a acın neresine eklenece ini belirlemek gerekmektedir. Belirleme yapıldıktan sonra eklenmi olan dü ümden itibaren köke olan yol üzerinde ilerleyerek denge ihlalinin ilk görüldü ü dü üm olan 20'ye z adı veriliyor. z'nin o yol üzerindeki torununa x ve arada kalan dü üm de y olarak isimlendiriliyor. Meydana gelen yol **zig-zag** biçiminde oldu undan y olarak adlandırılan dü ümün ekil 5.43 a)'da oldu u gibi sa a do ru döndürülmesi gerekmektedir. ekil 5.43 b)'de ise **zig-zig** durumunda ki a aç sola döndürülerek denge sa lanmalıdır.

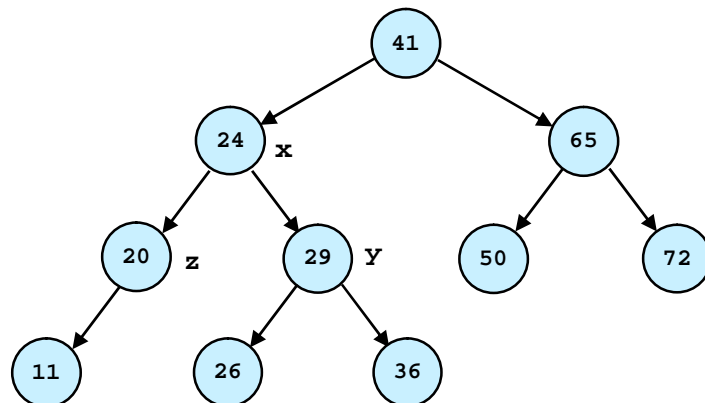


ekil 5.43 a) A aç ilk olarak sa a döndürülmelidir.



ekil 5.43 b) A aç son olarak sola döndürülmelidir.

ekil 5.44'te double rotation yöntemiyle ekleme yapılarak BST ve AVL özelli i korunmu olan a aç görülmektedir.



ekil 5.44 Double rotation uygulanmı AVL a acı.

Double rotation için `leftRightRotate` ve `rightLeftRotate` fonksiyonları a a daki gibi tanımlanmıştır. Fonksiyonların her ikisi de ihlalin görüldü ü z dü ümünü parametre olarak alıyor. A aç y dü ümünden itibaren sola, ardından da sa a döndürülüyor.

```
AVLTREE *leftRightRotate(AVLTREE *z) {
    z -> left = leftRotate(z -> left); // y dü ümü sola döndürülüyor
    /* Fonksiyondan geri dönen x dü ümü, z dü ümünün sol çocu u olarak
       atanıyor. Aynı zamanda y dü ümünün ebeveyni durumunda */
    return rightRotate(z); // z dü ümü bu defa sa a döndürülüyor
}
```

`rightLeftRotate` kodları da aynı mantıkla çalışmaktadır. Tamamen üstteki kodun simetriidir.

```
AVLTREE *rightLeftRotate(AVLTREE *z) {
    z -> right = rightRotate(z -> right); // y dü ümü sa a döndürülüyor
    /* Fonksiyondan geri dönen x dü ümü, z dü ümünün sa çocu u olarak
       atanıyor. Aynı zamanda y dü ümünün ebeveyni durumunda */
    return leftRotate(z); // z dü ümü bu defa sola döndürülüyor
}
```

Döndürme kodlarını yazdı mıza göre bir AVL ağacına ekleme işleminin fonksiyonunu da yazabiliriz. Fonksiyon parametre olarak eklenecek veriyi ve ağacın adresini almaktadır. Geri dönüş de erişilebilir şekilde gerekli döndürmeler uygulanarak AVL özelliğini sağlar yeni ağacın köküdür.

```
AVLTREE *insertToAVL(int x, AVLTREE *tree)
{
    if(tree != NULL)
    {
        if(x < tree -> data) // eklenecek verinin kökten büyüklü üne göre yaprağa
            tree -> left = insertToAVL(x, tree -> left); // kadar iniliyor
        else if(x > tree -> data)
            tree -> right = insertToAVL(x, tree -> right);
        else
            return tree; // Eklenen dü ümün adresi geri döndürülüyor

        /* Eklemeyen sonra her dü ümün yüksekliği güncelleniyor */
        tree -> height = maxValue(height(tree->left), height(tree->right)) + 1;

        /* E er balans faktör 1'den büyükse ve eklenen veri ağacın sol çocu unun
           datasından küçük ise ağaç sa a döndürülmelidir. */
        if((height(tree->left) - height(tree->right)) > 1 && x < tree->left->data)
            return RightRotate(tree);

        /* E er balans faktör 1'den büyükse ve eklenen veri ağacın sol çocu unun
           datasından büyük ise ağaç ilk olarak sola döndürülmeli, sonra da sa a
           döndürülmelidir. Yani leftRightRotate işlemi uygulanmalıdır. */
        if((height(tree->left) - height(tree->right)) > 1 && x > tree->left->data)
            return leftRightRotate(tree);

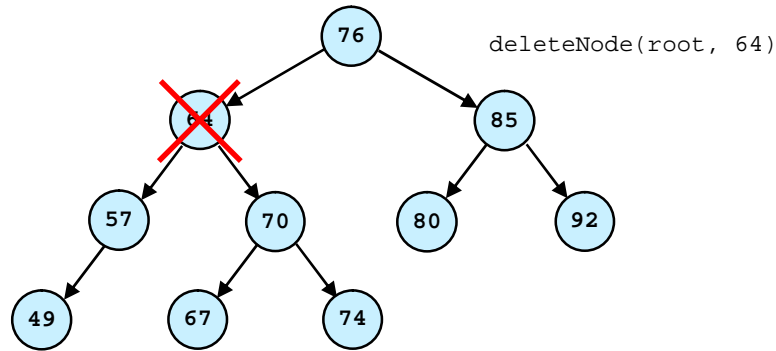
        /* E er balans faktör -1'den küçükse ve eklenen veri ağacın sa çocu unun
           datasından büyük ise ağaç sola döndürülmelidir. */
        if((height(tree->left) - height(tree->right)) < -1 && x > tree->right->data)
            return LeftRotate(tree);

        /* E er balans faktör -1'den küçükse ve eklenen veri ağacın sa çocu unun
           datasından küçük ise ağaç ilk olarak sa a döndürülmeli, sonra da sola
           döndürülmelidir. Yani rightLeftRotate işlemi uygulanmalıdır. */
        if((height(tree->left) - height(tree->right)) < -1 && x < tree->right->data)
            return rightLeftRotate(tree);
    }
    else
        tree = new_node(x);
    return tree;
}
```


AVL Aaçlarında Silme İlemi

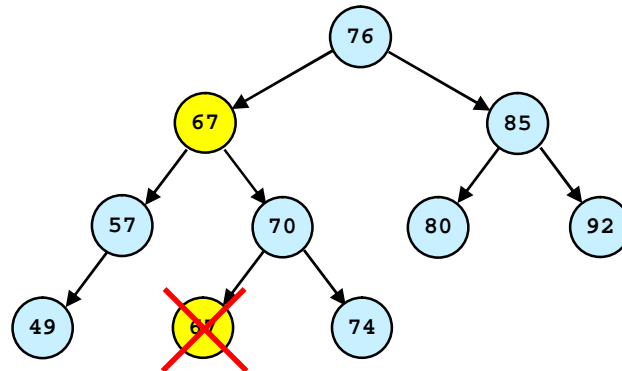
AVL ağaçlarında bir düğümü silme işlemi yapılırken hem denge özelliği, hem de BST özelliği kaybolmamalıdır. Silme işleminde eklemeye olduğu gibi bazı düğümlerin yükseklikleri de değişebilir ve tekrar denge durumuna getirmek gerekir. Bir düğüm eklendikten hemen sonra silindiğinde oluşan AVL ağacı, bir öncekinin aynısı değildir. Bir düğümü silmek istediğimizde eğer ağaçta hiç eleman yoksa fonksiyon NULL ile geri dönmelidir. Ağaçta eleman var ise, ancak aranan değer bir eşitlik varsa silme işlemi gerçekleştirilecektir. Silinecek düğüm ağacın çocuklarından herhangi biri ise ve verisi, üzerinde bulunduğu düğümün verisinden küçükse, sol tarafa doğru bir düğüm ilerleyip yeni bir kontrol yapılmalıdır. Büyükse bu defa sağ tarafta bir düğüm ilerleyip kontrolü tekrarlamamız gerekmektedir. Aranan düğüm bulunduğu yerde silme işleminde de 3 durumdan bahsedilebilir. Bunlar; eğer silinecek düğümün hiç çocuğu yoksa bu düğüm yapraklardan biridir ya da köktür. Düğüm direkt silinir ve ağacın denge durumu kontrol edilir. Denge bozulmuşsa gerekli döndürme işlemi yapılır. Silinecek düğümün bir çocuğu var ise, çocuk ebeveyninin yerine geçer ve ebeveyn silinerek denge kontrolü yapılır. Eğer silinecek düğümün iki çocuğu var ise sağ alt ağacındaki en küçük veriyi barındıran düğüm bulunarak silinecek düğüm kopyalanır ve kopyalanan sağ alt ağacındaki en küçük veriyi barındıran düğüm silinir. Denge bozulmuşsa gerekli döndürmeler yapılarak silme işlemi tamamlanmış olur. Bir örnekle konuyu daha iyi kavrayalım.

Aşağıda görülen ekil 5.45'teki AVL ağacından 64'ün silinmesi istendiğini kabul edelim.



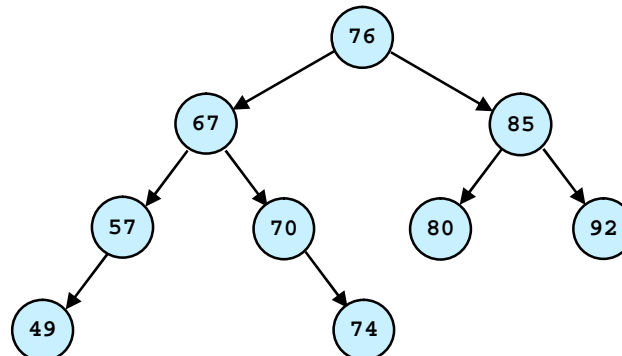
ekil 5.45 AVL ağacından 64 değeri silinmek isteniyor.

Düğümün iki çocuğunun olduğu görülüyor. Bu durumda sağ alt ağacında kendisine en yakın veriyi barındıran düğümü bulmamız gerekiyor. En yakın değer 67'dir ve bu düğüm silinmek istenen düğüm kopyalanmalıdır.



ekil 5.46 şimdi alt ağacındaki 67 değeri silinmelidir.

Sağ alt ağaçta yer alan 67 içeren düğüm silinmeli ve denge kontrolü yapılmalıdır. Silme işleminden sonra ağacın durumu ekil 5.47'de görülüyor.



ekil 5.47 Silme işleminden sonra ağacın son ekli.

A ağta denge bozulmadı için herhangi bir döndürme uygulanmamıdır. Eğer silinen alt ağdaki kopya düğümün de çocukları olsaydı, yukarıda konu girişi anlatmış kurallar tekrar uygulanacaktır. İmdi bir AVL ağından silme işlemi gerçekleştirilen deleteNode isimli fonksiyonu yazacağız fakat ilk olarak, deleteNode fonksiyonu içerisinde kullanacağımız sol alt ağ ile sağ alt ağ yüksekliklerinin farkını hesaplayan getBalance fonksiyonunu yazmamız gerekiyor.

```
int getBalance(AVLTREE* origin) {
    if (origin == NULL)
        return 0;
    return height(origin ->left) - height(origin ->right);
}
```

İmdi bir AVL ağından silme işlemi gerçekleştirilen deleteNode isimli fonksiyonu yazabiliriz.

```
AVLTREE *deleteNode(AVLTREE *root, int key) {
    if(root == NULL)
        return root;
    if(key < root -> data) // silinecek veri ebeveyninin verisinden küçükse
        root -> left = deleteNode(root -> left, key);
    else if(key > root -> data) // silinecek veri ebeveyninin verisinden büyükse
        root -> right = deleteNode(root -> right, key);
    else { // silinecek veri bulunmuş ise
        if((root -> left == NULL) || (root -> right == NULL)) {
            AVLTREE *temp = root -> left ? root -> left : root -> right;
            /* Eğer çocuk yoksa temp NULL de erini alıyor fakat bir çocuk varsa,
               çocuk temp'e atanıyor */
            if(temp == NULL) { // eğer silinecek düğümün hiç çocuğu yoksa
                temp = root;
                root = NULL;
            } else // eğer silinecek düğümün bir çocuğu varsa
                *root = *temp; // çocuk ebeveyninin yerine geçiyor
            free(temp); // temp siliniyor
        } else { // eğer silinecek düğümün iki çocuğu varsa
            AVLTREE *temp = minValue(root -> right);
            // sağ alt ağda en küçük veriye sahip olan düğüm bulunuyor
            root -> data = temp -> data;
            // bulunan minimum değer silinecek düğüm kopyalanıyor
            root -> right = deleteNode(root -> right, temp -> data);
            /* Artık silinecek değer temp'tedir. Fonksiyon kendisini sağ çocuğu ve
               silinecek temp de eriyile tekrar çağırıyor */
        }
    }
    if (root == NULL)
        return root;
    // Ağın yüksekliği güncelleniyor
    root -> height = max(height(root -> left), height(root -> right)) + 1;
    // Eğer balans 1'den büyükse ve sol alt ağın balansı 0'dan büyük veya eşitse
    if (getBalance(root) > 1 && getBalance(root -> left) >= 0)
        return rightRotate(root); // sağa döndür
    // Eğer balans 1'den büyükse ve sol alt ağın balansı 0'dan küçükse
    if (getBalance(root) > 1 && getBalance(root -> left) < 0) {
        root -> left = leftRotate(root -> left); // ilk önce sola döndür
        return rightRotate(root); // sonra sağa döndür
    }
    // Eğer balans -1'den küçükse ve sağ alt ağın balansı 0'dan küçük veya eşitse
    if (getBalance(root) < -1 && getBalance(root -> right) <= 0)
        return leftRotate(root); // sola döndür
    // Eğer balans -1'den küçükse ve sağ alt ağın balansı 0'dan büyükse
    if (getBalance(root) < -1 && getBalance(root -> right) > 0) {
        root -> right = rightRotate(root -> right); // ilk olarak sağa döndür
        return leftRotate(root); // sonra sola döndür
    }
    return root;
}
```

5.1 ÖNCEL KL KUYRUKLAR (Priority Queues)

Öncelikli kuyruklar (*priority queues*), terim anlamıyla gündelik ya amda sık kar ıla tı ımız bir olguyu belirler. Bazı durumlarda bir i i öteki i lerin hepsinden önce yapmak zorunda kalabiliriz. Örne in, bir fatura ödeme veznesinde kuyru a girenler arasında öncelik sırası önde olanındır. Ancak, bir kav akta geçi önceli i cankurtaranındır. Bir hava meydanına ini sırası bekleyen uçaklar arasında, öncelik sırası acil ini isteyen uça ındır. Bir hastanede muayene önceli i ise durumu en acil olan hastanıdır.

Bilgisayarlarda öncelikli kuyrukların kullanımına örnek olarak printer ve i letim sistemi verilebilir. A yazıcılarında'da az sayfaya sahip belge önce yazılır. Çok sayfası olan bir belge yazdırılmak üzere yazıcıya gönderilmi ve hemen akabinde iki sayfalık bir belge daha yazdırılmak istenmi olabilir. ki sayfalık belgenin çıktısını almak için sayfalar dolusu belgenin yazdırma i leminin bitmesi beklenmez. Arada bir bölümde iki sayfa yazdırılır. letim sisteminde ise en kısa zamanlı i lem önce çalı ır. Gerçek zamanlı i lemler de öncelik de erine sahiptir.

Görüldü ü gibi, bir koleksiyon içinde öncelik sırasını farklı amaçlarla belirleyebiliriz. En basiti, ilk gelen ilk çıkar dedi imiz FIFO (*first in first out*) yapısıdır. Ama bu yapı kar ıla ıllacak bütün olasılıklara çözüm getiremez. Dolayısıyla, koleksiyonun ö elerini istenen önceli e göre sıralayan bir yapıya gereksinim vardır. Biraz genellemeyle, öncelikli kuyruklar yapısına da kuyruk diyece iz; ama burada yüklenen anlamı FIFO yapısından farklı olabilir. Son giren de ilk çıkabilir, ilk giren de ilk çıkabilir. Duruma göre de i mektedir. Önceli i en yüksek olan kuyruktan ilk çıkar. Soyut bir veri tipi (ADT – *Abstract Data Type*) olarak öncelikli kuyruklarda sadece sayılar ya da karakterler de il, karma ık yapılar da olabilir. Örnek olarak bir telefon rehberi listesi, soyisim, isim, adres ve telefon numarası gibi elemanlardan oluşmaktadır ve soyisme göre sıralıdır. Öncelikli kuyruklardaki elemanların sıralarının elemanların alanlarından birisine göre olması da gerekmez. Elemanın kuyru a eklenme zamanı gibi elemanların alanları ile ilgili olmayan dı sal bir de ere göre de sıralı olabilirler.

Öncelikli kuyruklar, temel kuyruk i lemlerinin sonuçlarını elemanların gerçek sırasının belirledi i veri yapılarıdır. Azalan ve artan sırada olmak üzere iki tür öncelik kuyru u vardır. Artan öncelik kuyruklarında (*min heap*) elemanlar herhangi bir yere eklenebilir ama sadece en küçük eleman çıkarılabilir. Azalan öncelik kuyru u (*max heap*) ise artan öncelik kuyru unun tam tersidir. Artan öncelik kuyru unda önce en küçük eleman, sonra ikinci küçük eleman sırayla çıkarılacaktır. Birkaç eleman çıkarıldıktan sonra ise daha küçük bir eleman eklenirse do al olarak kuyruktan çıkarıldı ında önceki elemanlardan daha küçük bir eleman çıkımı olacaktır.

Kuyruktaki her nesnenin kar ıla tırılabilir bir öncelik de eri (*key*) vardır. Bazı sistemlerde küçük de er önceliklidir, bazı sistemlerde de küçük de erler, küçük öncelikli olabilir. lk nesne kuyru un ba ına konulur ve önceli i belirten bir key de eri atanır. Öncelikli kuyruklarda da di er kuyruklarda oldu u gibi ekleme ve silme i lemleri mevcuttur.

Bu kuyrukları bilgisayarda nasıl implemente edebiliriz? Daha önceki kuyrukları ba lı listelerle ve dizilerle implemente etmi tik. Peki, burada ne yapmalıyız? Ba lı listelerde i lem oldukça kolaydı ve listenin sonuna ekleyip, ba tan çıkarma i lemi yapıyordu. Burada ise öncelik durumu varsa ba a eklenmeli, küçük öncelikli ise araya eklenmelidir. Numarasına göre farklı yerlere atanmalı ya da önceli i yüksek olan kuyruktan çıkmalıdır.

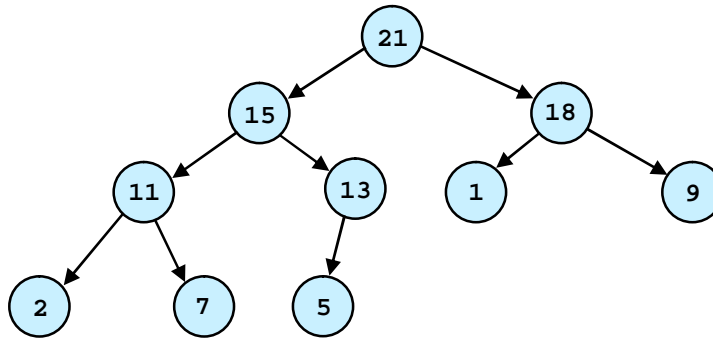
Tüm bu kuralların uygulandı ı bir dizi kullandı ımızı varsayalım ve dizinin içindeki de erlerin de öncelikli de erler (*bu öncelikli de er de küçük de er olsun*) oldu unu kabul edelim. Bu durumda en yüksek öncelikli de eri kuyruktan çıkarmak için dizi üzerinde arama yapmak gerekmektedir. Bir dizi üzerindeki en küçük de eri bulmak için bir `for` döngüsü kullanılabilir fakat dizi içerisinde 1 milyon tane veri varsa elbette bunun maliyeti de yüksek olacaktır ve 1 milyon tane iterasyon gereklidir. O halde minimum elemanı bulmak için daha hızlı, daha efektif farklı bir veri yapısı kullanılmalıdır.

Binary Heap (kili Yı ın)

Yı ın (*heap*) yapısı n tane elemandan oluş an ve öncelikli kuyrukları gerçekle tirmek için kullanılan ikili a aç ekinde bir yapıdır. kili arama a acı, arama i lemini hızlı yapmak için üretilmi bir veri yapısı iken yı ın, bir grup eleman arasından önceli i en yüksek olan elemanı hızlı bulmak ve bu elemanı yapıyı bozmadan hızlıca silmek için üretilmi bir veri yapısıdır. kili arama a acında her dü üm sol alt a acındaki tüm dü ümlerden büyük, sa alt a acındaki tüm dü ümlerden küçük ya da e it iken, yı ında her dü üm sol ve sa alt a acındaki tüm dü ümlerden büyüktür ya da küçüktür. Sol ve sa alt a aç arasında büyüklük küçüklük ili kisi açısından bir zorlama yoktur. Ayrıca ikili arama a acında dü ümlerin sol veya sa çocukları olabilece i gibi hiç çocu u olmayabilir. Yı ında ise elemanlar seviye seviye yerle tirilir. Üstteki seviye dolmadan alttaki seviyede eleman bulunmaz. Dolayısıyla son seviyeden önceki seviyedeki tüm dü ümlerin iki çocu u vardır. Son satır dolu olmayabilir fakat soldan sa a do ru doldurulmalıdır.

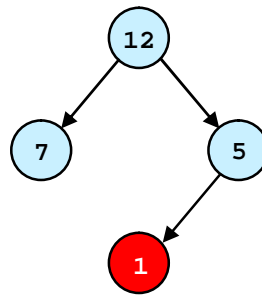
Yı ınlarda en büyük de erli eleman a acın kökündeyse bu dizilime **max-heap**, en küçük de erli eleman a acın kökündeyse bu dizilime **min-heap** denir. Max heap'te her bir dü ümün de eri, çocuklarının de erlerinden küçük olamaz. Büyüktür ya da e ittir. Min heap'te ise her bir dü ümün de eri, çocuklarının de erlerinden küçüktür veya e ittir, büyük olamaz. Örne in be sayfalık bir belgeyi yazıcıya gönderdi imizde bir öncelik de eri atanacaktır. Ardından be sayfalık ba ka bir belge daha yazıcıya gönderildi inde bu belgeler de ilk belgelerin öncelik de erine sahip olacaktır, yani öncelik de erleri e ittir.

ekil 5.48'de görülen max-heap a aç ikili bir a açtır (**BT**-binary tree) fakat ikili arama a aç (**BST**-binary search tree) de ildir. Her dü üm kendi çocuklarından büyüktür.



ekil 5.48 Max-Heap a aç.

A a rdaki a aç ise bir heap de ildir. Çünkü her seviye dolu olmasına kar ın, son seviyenin doldurulmasına en soldan ba lanmamı tr.



ekil 5.49 Heap özelli i olmayan bir a aç.

Heap, bir a açla temsil edilebilir. A aç da do rusal olmayan ba lı bir listedir. Heap, liste ekinde yapılabilir fakat listelerle yönetmek yerine dizilerle i lem yapmak daha kolaydır. Çünkü her seviye doludur ve son satır da soldan sa a do ru bir dizilime sahiptir. Bu nedenle y ınlar, genelde tek boyutlu dizilerle implemente edilirler.

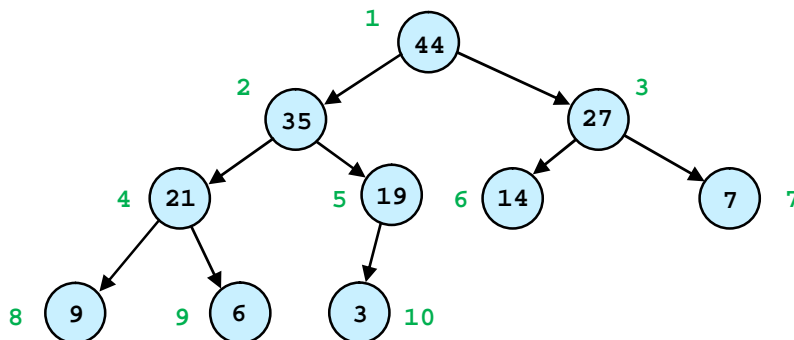
Mapping (E leme): A açın öncelikle kökü dizideki ilk elemandır. Heap, bir dizide tam ikili a aç yapısı kullanılarak gerçekleştirildi inde, dizinin 1. elemanından ba lar, 0. indis kullanılmaz. Dizi sıralı de ildir, yalnız 1. indisteki elemanın en öncelikli (*ilk alınacak*) eleman oldu u garanti edilir. Dizinin bir i indisindeki elemanın ebeveyninin (*parent*) indisi $i/2$ 'ye e ittir.

$$\text{parent}(i) = i/2$$

Buradaki kö eli paranteze benzeyen simge, içindeki ondalıklı sayısal de eri a a ı yuvarlamaktadır. Örne in dizinin 4 ve 5. indisindeki elemanların ebeveynlerinin indisi $4/2 = 2$ ve $5/2 = 2$ 'dir. Öyleyse 4. indisteki eleman sol çocuk, 5. indisteki eleman ise sa çocuk. Tersten dü ünürsek kökün indisi 1 ise, sol çocu unun indisi $2i$, sa çocu un indisi $2i + 1$ 'dir.

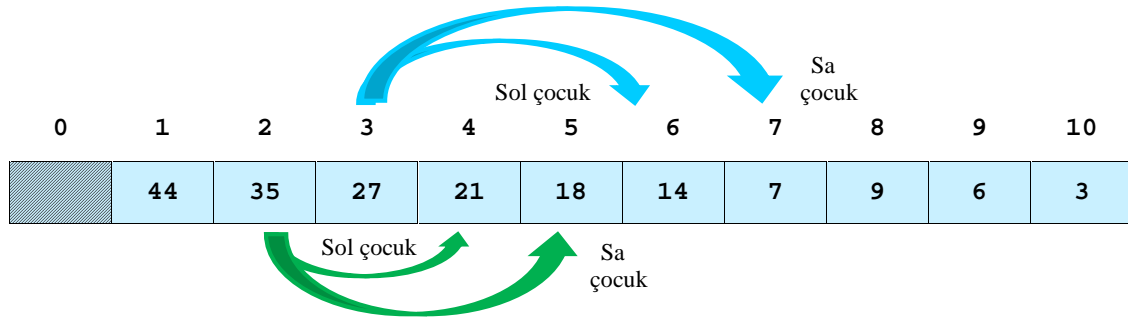
$$\begin{aligned} \text{left}(i) &= 2i \\ \text{right}(i) &= 2i + 1 \end{aligned}$$

imdi heap'in dizi implementasyonunu ekil 5.50 üzerinde inceleyelim. A açtaki elemanların hepsini bir dizide saklayaca ız. A açta 10 eleman oldu una göre dizi boyutu 11 olmalıdır. Çünkü 0 indisli dizinin ilk elemanın kullanılmadı nı söylemi tik. Dolayısıyla veri ataması yapılmayacaktır.



ekil 5.50 Örnek heap a aç.

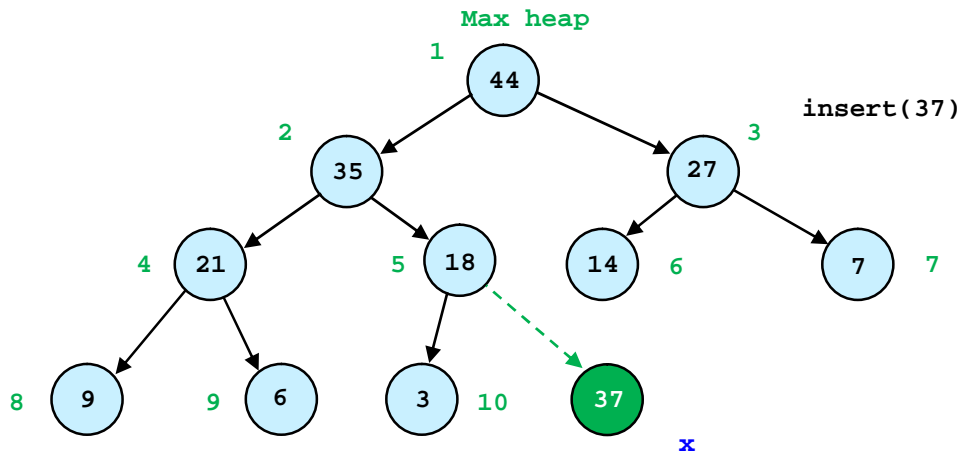
Heap'in dizi ile implemente edilmiş hali ekil 5.51'de görülmektedir. Dizide 2. indiste bulunan elemanın sol çocuğu 4. indisteki 21, sağ çocuğu ise 5. indisteki 18'dir. Yine aynı şekilde 3 numaralı indisteki elemanın sol çocuğu 6. indiste yer alan 14 ve sağ çocuğu da 7. indisteki 7'dir.



ekil 5.51 Heap'in dizi uygulaması.

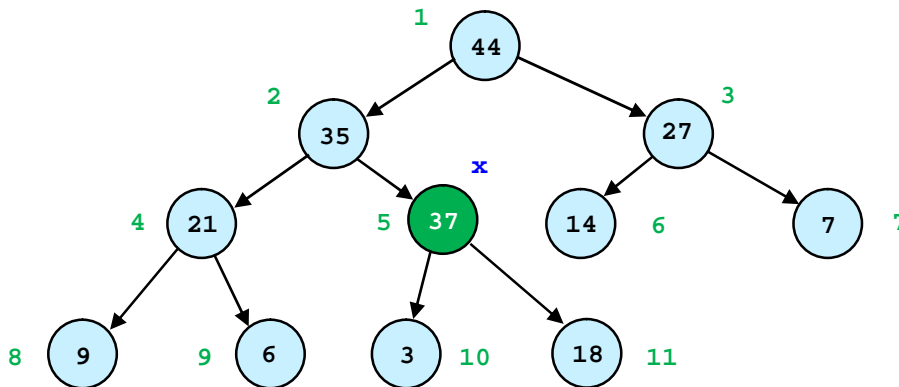
Heap İşlemleri

1- Insert (Ekleme): Eklenmesi gereken bir x değeri heap'in en alt satırında ilk boşluğa yerleştirilir. Sonra ebeveyninin değeriyle karşılaştırılır. Eğer eklenmesi gereken x değeri ebeveyninin değerinden büyük ise yerleri değiştirilir (*swap*) ve tekrar ebeveyninden büyük mü diye bakılır. Bu işlem eklenmesi gereken verinin ebeveyninden büyük olmadığı duruma kadar devam eder. Örneğin heap'e 37 sayısı eklenmesi isteniyor. Aşağıdaki şekilde 37 sayısının heap'e eklenmesi adım adım gösterilmiştir. Ekil 5.52 a)'da görüldüğü gibi, eklenmesi gereken bu değer x olarak adlandırılır. Heap'te soldan sağa bir yerleşim olduğu için x , 3'ün yanına atanmalıdır.



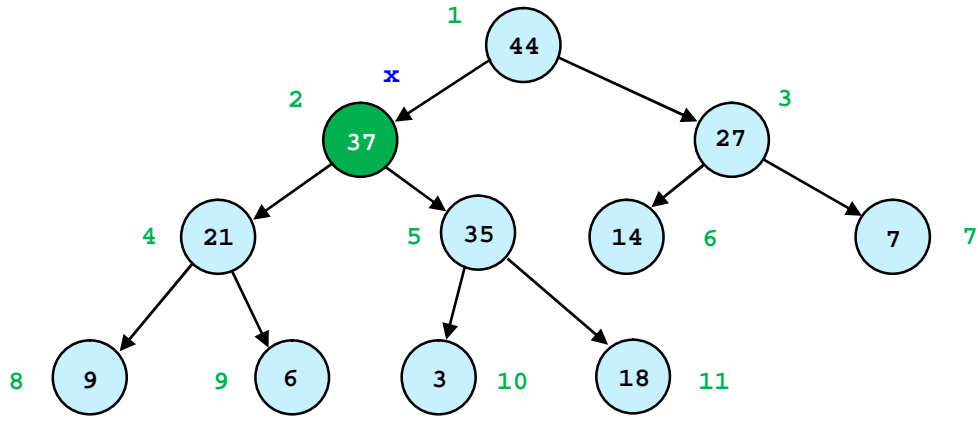
ekil 5.52 a) Heap'e 37 sayısı ekleniyor.

Atandıktan sonra ebeveyni olan 18 ile karşılaştırılır ve büyük olduğu için *swap* (yer değiştirme) işlemi yapılır. Bu durum ekil 5.52 b)'de görülmektedir. x bu defa ebeveyni olan 35 ile karşılaştırılır ve hala ebeveyninden büyük olduğu için tekrar *swap* işlemi yapılır.



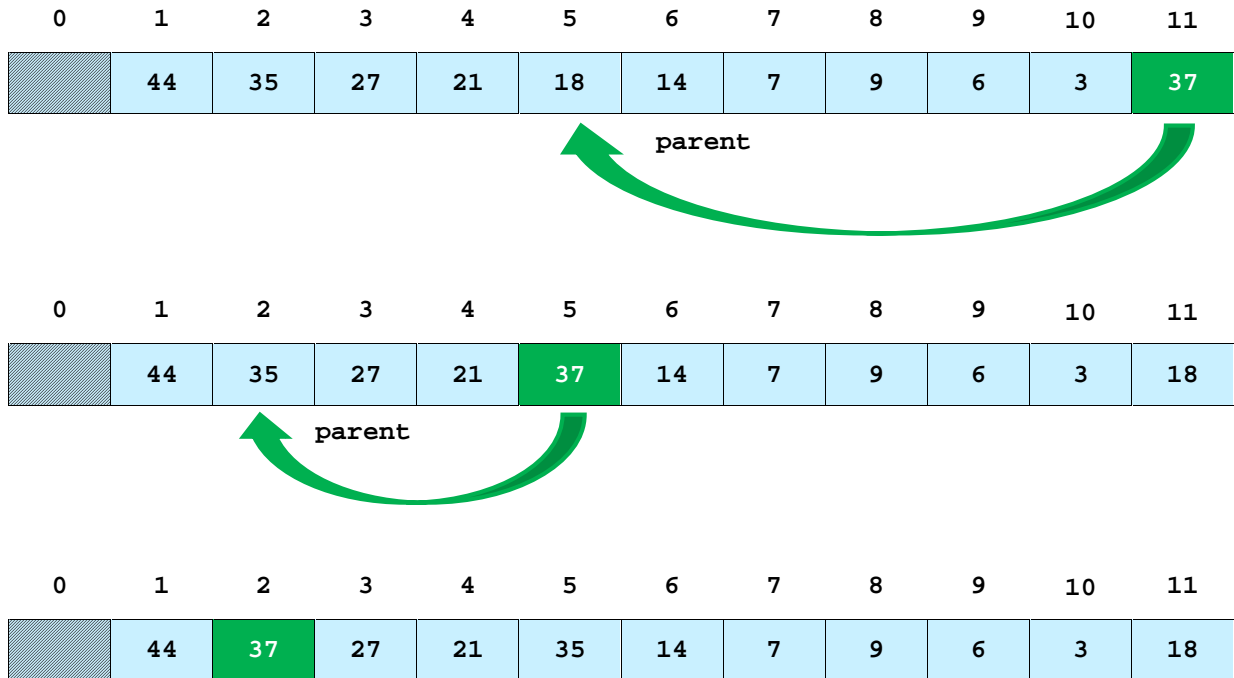
ekil 5.52 b) 37 sayısı ebeveyniyle yer değiştiriyor.

x artık ebeveyni olan 44'ten büyük olmadığı için işlem sona erer. Ekleme işleminden sonra heap'e 37 sayısının son durumu ekil 5.52 c)'de görülmüştür.



ekil 5.52 c) 37 sayısı bir kez daha ebeveyniyle yer de i tiriyor.

Burada yapılan i lemlere **yukarıya do ru tırmanma** (*percolate up*) denir. Ekleme i leminin dizi uygulaması ekil 5.53'te gösterilmi tir.



ekil 5.53 37 sayısı heap'e ekleniyor. Ardından ebeveyniyle kar ıla tırılıyor ve e er ebeveyninden büyükse yer de i tiriyor.

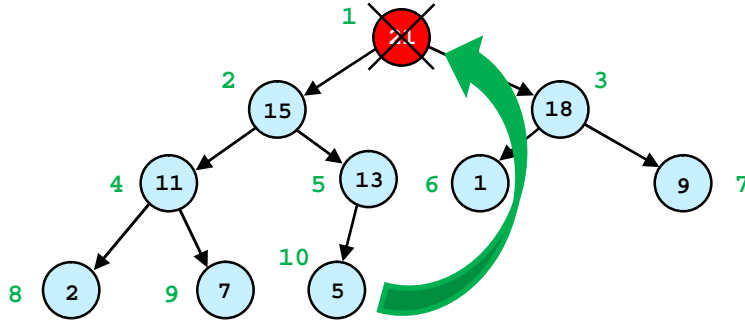
Tanımlanmış bir max heap dizisine de er ekleyen `insertToMax_heap` fonksiyonu aşağıdaki gibi yazılabilir. Fonksiyon parametre olarak eklenecek heap'in adresini, eklenecek de eri ve heap'in sıradaki bo indeksini almaktadır. Geriye bir de er döndürmeyece i için türü `void` olarak tanımlanmıştır. Fonksiyon içerisinde çağ rılan `swap` fonksiyonunu artık biliyorsunuz. `swap()`, kendisine gelen iki de i kenin de erlerini birbirleriyle de i tirmektedir.

```
void insertToMax_heap(int *array, int x, int index) {
    if(index == SIZE)
        printf("Heap dolu !\n");
    else {
        array[index] = x;
        while(index != 1 && array[index / 2] < array[index]) {
            swap(&array[index / 2], &array[index]); // yer de iştiriliyor
            index /= 2;
        }
    }
}
```

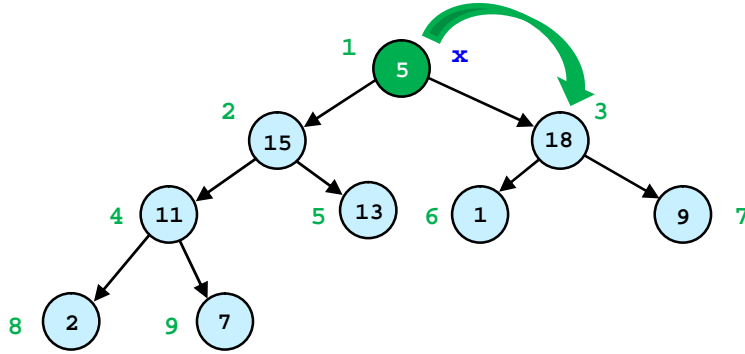
Aynı fonksiyon ufak bir oynama ile min heap için de yazılabilir. `while` ko ulu içerisindeki küçüktür (<) i aretini büyüktür (>) i aretine çevirmek ve fonksiyonun adını de i tirmek yeterlidir.

2- Delete (Silme): Bir max heap'ten maksimum eleman silindi inde bir boşluk oluşmaktadır. Dizi boşluk olmayacak biçimde düzenlenmeli, “tam ağaç” yapısını bozmamak için en sondaki eleman uygun bir konuma kaydırılmalıdır. Bu kaydırma yapılırken maksimum yığın yapısının da korunmasına dikkat edilmelidir. Ağacın kökündeki eleman silinir. Yerine ağacın en alt satırının en sağındaki x olarak adlandırılan düğüm konulur. Eleman x çocuklarının değerlerinden küçük ise en büyüğü ile swap yapılır. Bu işlem x çocuklarından küçük olmayıncaya kadar devam eder.

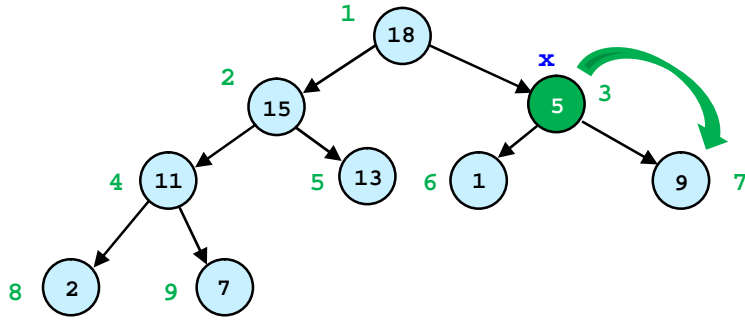
ekil 5.54'te silme işleminin basamakları gösterilmiştir. Silme fonksiyonu 21 değerini geri döndürür ve heap'ten siler. Yerine son satırının en sağındaki veriyi atar ve çocuklarıyla teker teker karşılaştırılır. Eleman küçükse en büyüğüyle yer değiştirilir ve bu işlem küçük olmayıncaya kadar devam eder.



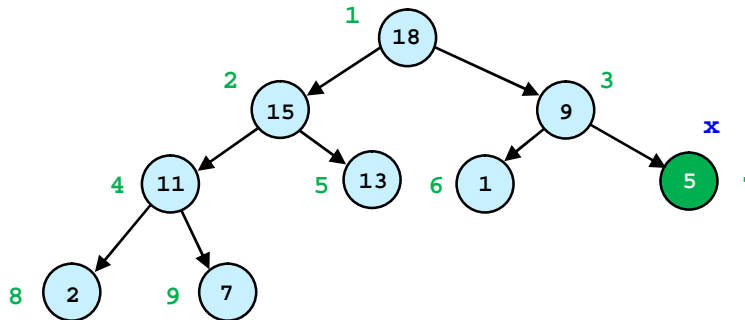
ekil 5.54 a) 21 sayısı heap'ten siliniyor.



ekil 5.54 b) Silinen 21 sayısının yerine en alt satırın sağındaki 5 değeri atanıyor.



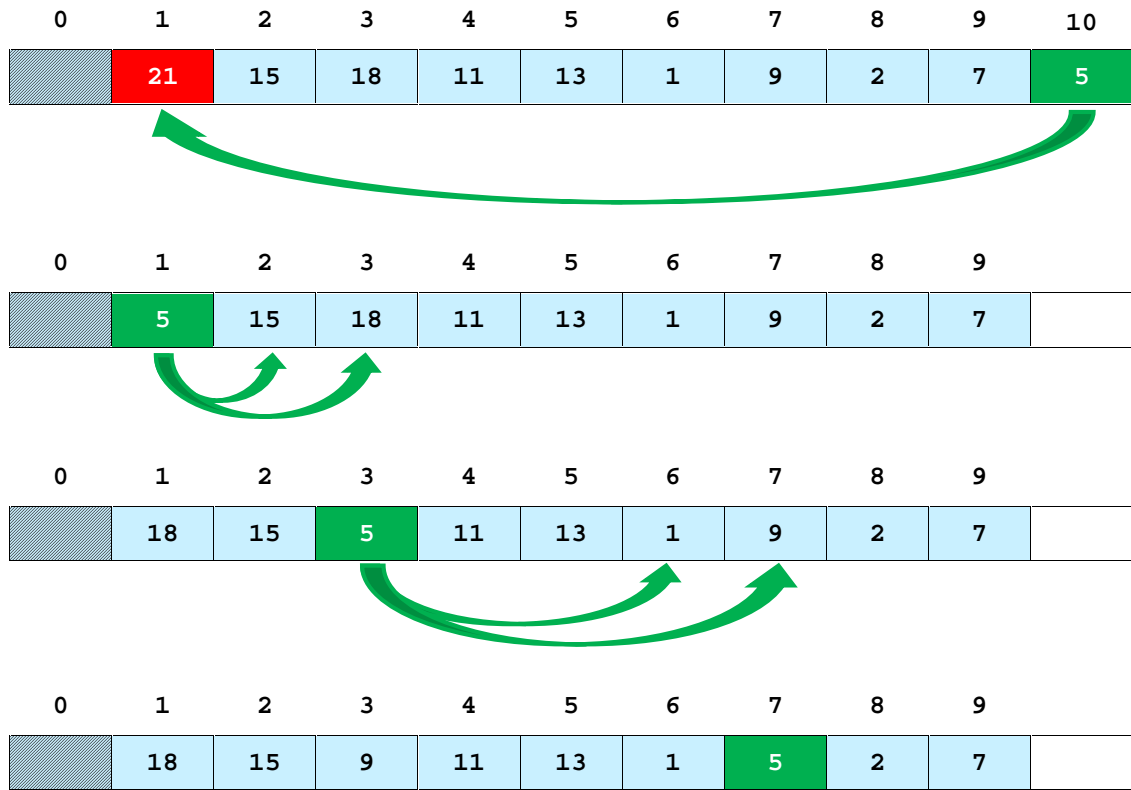
ekil 5.54 c) 5 değeri çocuklarıyla karşılaştırılıyor ve büyük olanla yer değiştiriliyor.



ekil 5.54 d) Silme işleminden sonra ağacın son durumu.

Görüldüğü gibi Max heap özelliğinde maksimum eleman her zaman köktedir. Her silme işlemi yapıldığı zaman kuyruktaki maksimum eleman köke gelir ve her silme işlemi maksimumuma bir keredeye ulaşma imkânı verir. Burada ağacı tekrar

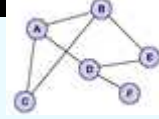
düzenlemenin elbette bir maliyeti vardır. Kökten ba layarak en alttaki yapra a kadar bir dola ma gerektirir ve bu dola ma da a acın yüksekli i ile orantılıdır. Peki n dü üme sahip bir heap'in yüksekli i nedir? Yükseklik $\lg n$ civarındadır. Örne in 1 milyon tane verisi olan heap'te silme i leminden sonra en fazla 20 iterasyon yapmak yeterlidir. Silme i leminde max heap'deki yukarı tırmanma i leminin tersi bir durum vardır. Bu i leme **percolate down** denmektedir. Yapılan delete i leminin dizi uygulaması ekil 5.55'te görölmektedir.



ekil 5.55 Silme i leminin dizi uygulaması.

imdi max heap'ten bir elemanı silen deleteMax isimli fonksiyonu yazalım. Fonksiyon parametre olarak, elemanı silinecek heap'in adresini ve heap'in ekleme yapılabilecek bo indeksini almaktadır. Silinen elemanın de eriyle de geri dönmelidir.

```
int deleteMax(int *array, int index) {
    int max, i = 1;
    if(index == 1) {
        printf("Heap bos...\n");
        return 0;
    } else {
        max = array[i];
        array[i] = array[index-1]; // son eleman başa atanarak max eleman siliniyor
        array[index-1] = 0; // heap'in en başa atanan son elemanı siliniyor
        while(array[i] < array[2*i] || array[i] < array[(2*i) + 1]) {
            if(array[2*i] > array[(2*i) + 1]) { // sol çocuk büyükse
                swap(&array[2*i], &array[i]); // sol çocukla yer de iştiriliyor
                i = 2*i;
            }
            else { // sa çocuk büyükse
                swap(&array[(2*i) + 1], &array[i]); // sa çocukla yer de iştiriliyor
                i = (2*i) + 1;
            }
        }
        return max;
    }
}
```

BÖLÜM

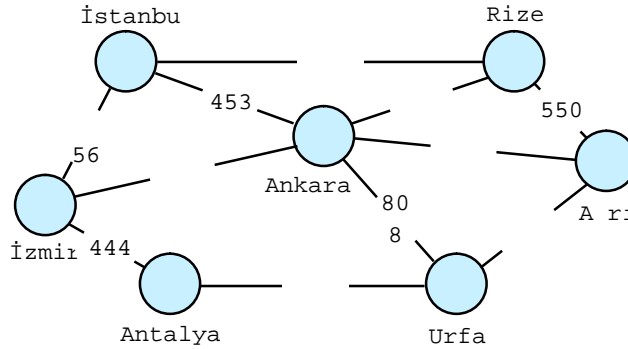
Graphs (Çizgeler)

6

6.1 G R

Çizge kuramının ba langıcı Königsberg'in 7 köprüsü (*Kaliningrad/Rusya*) problemine dayanır. 1736'da Leonhard Euler'ın söz konusu probleme ili kin kullandı ı sembol ve çizimler graf kuramına ili kin ba langıç olarak kabul edilir. Problem, "Pregel nehri üzerinde iki ada birbirine bir köprü ile, adalardan biri her iki kıyıya iki er köprü, di eri de her iki kıyıya birer köprü ile ba lıdır. Her hangi bir kara parçasında ba layan ve her köprüden bir kez geçerek ba langıç noktasına dönülen bir yürüyü (*walking*) var mıdır?" sorusunu içeriyordu. Euler kara parçalarını dü üm, köprüleri de kenar kabul eden bir çizge elde etti. Söz konusu yürüyü ün ancak her dü ümün derecesinin çift olması durumunda yapılabilece ini gösterdi. Graf kuramının bu ve buna benzer problemlerle ba ladı ı dü ünülür. Günlük ya amdaki birçok problemi, graf kuramı uzayına ta yarak çözebiliyoruz.

Graf (*çizge*), bilgisayar dünyasında ve gerçek hayatta çe itli sebeplerle kar ıla ılan bir olay veya ifadenin dü üm ve çizgiler kullanılarak gösterilmesi eklidir. Fizik, kimya gibi temel bilimlerde, mühendislik uygulamalarında ve tıp biliminde pek çok problemin çözümü ve modellenmesi graflara dayandırılarak yapılmaktadır. Örne in elektronik devreleri (*baskı devre kartları*), entegre devreleri, ula ım a larını, otoyol a mını, havayolu a mını, bilgisayar a larını, lokal alan a larını, interneti, veri tabanlarını, bir haritayı veya bir karar a acını graflar kullanarak temsil etmek mümkündür. Ayrıca planlama projelerinde, sosyal alanlarda, kimyasal bile iklerin moleküler yapılarının ara tırılmasında ve di er pek çok alanda kullanılmaktadır. Örne in Türkiye'de bazı illerin karayolu ba lantıları ile yolun uzunlu unu gösteren a a ıdaki yapı, yönsüz bir graftır.



ekil 6.1 Yönsüz graf örne i.

Uygulamada karayolları ve havayolları rotaları farklıdır. Bu yüzden havayollarında yönlü graf kullanılması daha mantıklıdır. E er kullanımda bir simetri varsa (*gidi ve geli yolu aynı gibi*) yönsüz graf kullanmak daha iyidir.

ekil 6.1'de verilen yapıyı grafın teknik deyimleri ile belirtti imizde iki il arasındaki yollar **ayrıt** (*edge*), iki ayrıtın birle ti i yer **tepe** ya da **dü üm** (*vertex, node*) olarak anılır. Biz bu derste herhangi bir tepeyi (*dü üm*) küçük harf ve o tepenin indisi ile örne in v_4 biçiminde gösterece iz. Grafı olu turan tepelerin sırası önemli olmaksızın yalnız bir kez yazıldı ı kümeyi de (*set*) büyük harf V ile belirtece iz. Örne in yukarıdaki grafta V 'nin elemanı olan tepeler (*vertices*);

v_1	İstanbul	v_5	Urfa
v_2	İzmir	v_6	Rize
v_3	Ankara	v_7	A r ı
v_4	Antalya		

olarak belirlenirse,

$$V(G) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \text{ olur.}$$

Tek bir ayrıtı küçük harf e_i ile, ayrıtların olduğu kümeyi (*set*) ise büyük harf E ile belirteceğiz. Her tepe bir bilgi parçasını gösterir ve her ayrıtı iki bilgi arasındaki ilişkiyi gösterir. Eğer ayrıtlar üzerinde bir yönlülük belirtilmemişse bu tür graflar **yönsüz graf** (*undirected graph*) adını alırlar. Yönsüz grafın ayrıtları bir parantez çifti içinde o ayrıtı birleştirici tepeler yazılarak belirtilir. Örneğin (v_3, v_6) ayrıtı, ANKARA ile RZE arasındaki yolu belirtmektedir. Yönsüz gratta ayrıtı belirten tepelerin sırasının değiştirilmesi bir değişiklik oluşturmaz. Öteki deyişle (v_4, v_6) ile (v_6, v_4) aynı ayrıttır. Ayrıca veri yapısının kolay kurulumu sağlamak ve veri yapısını kurarken gözden kaçan bir ayrıtı olmasını daha kolay denetlemek için indisleri daha küçük olan tepe önce yazılır. Bu yazım biçimine göre yukarıdaki örnekte ayrıtı listesi şöyledir;

$$E = \{ v_1, v_2, 564, v_1, v_3, 453, v_1, v_6, 1141, v_2, v_3, 579, v_2, v_4, 444, v_3, v_5, 808, v_3, v_6, 820, v_3, v_7, 1054, v_4, v_5, 906, v_5, v_7, 617, v_6, v_7, 550 \}$$

Buradaki 564, 453, ... gibi sayılar, iki şehir arasındaki uzaklık, maliyet ya da iki router arasındaki trafik, bant genişliğini belirten bir ayrıttır.

Aynen ağaç gibi graflar da doğrusal olmayan veri yapıları grubuna girerler. Başlı listeler ve ağaç grafların özel örneklerindedir. Bir gratta düğümler dairelerle, ayrıtlar da çizgilerle gösterilir.

$$\begin{aligned} V &= \{v_0, v_1, v_2, v_3, v_4, \dots, v_{n-1}, v_n\} \text{ Tepeler (vertices)} \\ E &= \{e_0, e_1, e_2, e_3, e_4, \dots, e_{m-1}, e_m\} \text{ Ayrıtlar (edges)} \\ G &= \{V, E\} \text{ Graf (graph)} \end{aligned}$$

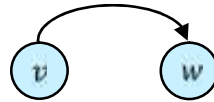
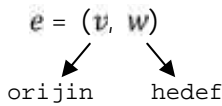
Bu anlatılanlardan sonra bilimsel olarak grafi şu şekilde tanımlayacağız; Bir G grafi, tepeler olarak adlandırılan boş olmayan bir $V(G)$ sonlu nesnelere kümesi ile G 'nin farklı tepe çiftlerinin düzensiz sıralanmış ve V 'nin tepelerini birleştirici bir $E(G)$ (*boş olabilir*) ayrıtlar kümesinden oluşur ve $G = (V, E)$ şeklinde gösterilir. Buradaki ayrıtlar G 'nin ayrıtları diye adlandırılır. G grafının tepeler kümesi $V(G) = v_1, v_2, \dots, v_n$ 'nin eleman sayısına G 'nin sıralanmış (*order*) denir ve $|V(G)| = n$ olarak gösterilir. Diğer taraftan ayrıtlar kümesi $E(G) = e_0, e_1, \dots, e_m$ 'nin eleman sayısına boyut (*size*) denir ve $|E(G)| = m$ olarak gösterilir.

Grafları ayrıtların yönlü olup olmamasına göre **yönlü graflar** ve **yönsüz graflar** olarak ikiye ayırmak mümkündür. Ayrıca ayrıtların ağırlıklı olmasına göre **ağırlıklı graflar** veya **ağırlıksız graflar** isimleri verilebilir.

Terminoloji, Temel Tanımlar ve Kavramlar

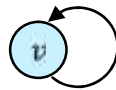
Yönsüz Ayrıtı (Undirected Edge): Çizgi şeklinde yönü belirtilmeyen ayrıtlar yönsüz ayrıtlardır. Sırasız node çiftleriyle ifade edilir. (v, w) ile (w, v) olması arasında fark yoktur.

Yönlü Ayrıtı (Directed Edge / digraph): Ok şeklinde gösterilen ayrıtlar yönlü ayrıtlardır. Sıralı node çiftleriyle ifade edilir. Birinci node **orijin**, ikinci node ise **hedef** olarak adlandırılır. Örneğin ekil 6.2'de (v, w) ile (w, v) aynı değildir.



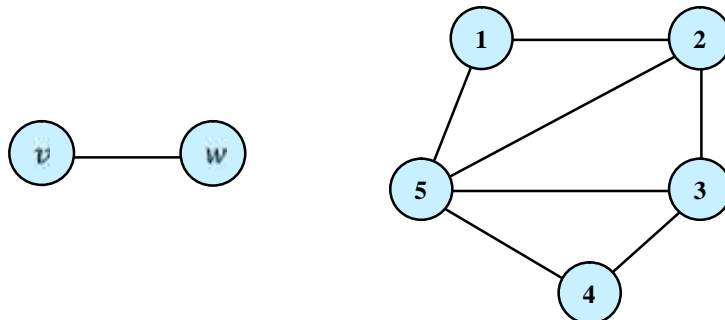
ekil 6.2 Sıralı tepe çifti.

Self Ayrıtı (Döngü –Loop): (v, v) şeklinde gösterilen ve bir tepayı kendine bağlayan ayrıttır.



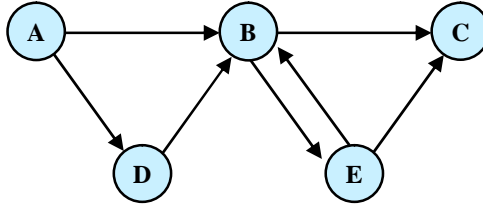
ekil 6.3 Self ayrıtı.

Yönsüz Graf (Undirected Graph): Tüm ayrıtları yönsüz olan grafa yönsüz graf denilir. Yönsüz gratta bir tepe çifti arasında en fazla bir ayrıtı olabilir. e sırasız bir çifttir. Bir ayrıtı çoklu kopyalarına izin verilmez. Ağaçlar bir graftır fakat her graf bir ağaç değildir. $e = v, w = (w, v)$



ekil 6.4 Sırasız çift ve yönsüz graf.

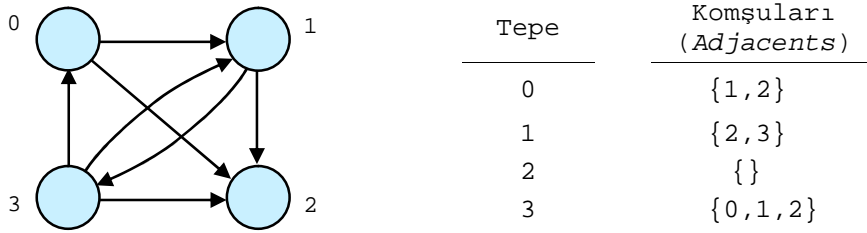
Yönlü Graf (Directed Graph, Digraph): Tüm ayrıtları yönlü olan grafa yönlü graf adı verilir. Yönlü grafa bir tepe çifti arasında ters yönlerde olmak üzere en fazla iki ayrıtlı olabilir. e sıralı bir çifttir. Her e ayrıtlı bazı v tepesinden di er bazı w tepesine yönlendirilmiştir. Yönsüz graflar, yönlü graf olabilir ama yönlü graf yönsüz graf olamaz.



ekil 6.5 Yönlü graf.

Yönlü graf tek yönde hareketli oldu u halde yönsüz graf her iki yönde de hareket eder,

Kom u Tepeler (Adjacent) : Aralarında do rudan ba lantı (ayrıtlı) bulunan v ve w tepeleri kom udur. Di er tepe çiftleri kom u de ildir. e ayrıtlı hem v hem de w tepeleriyle bitiktir (incident) denilir.

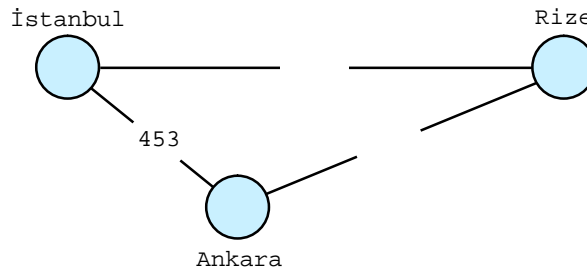


ekil 6.6 Kom uluk tablosu (adjacency table).

Kom uluk ve Biti iklik: Bir G grafi kom uluk ili kisiyle gösteriliyorsa $G_{vv} = v_i, v_j \dots$ biti iklik ili kisiyle gösteriliyorsa $G_{ve} = v_i, e_j \dots$ ekinde yazılır. (G_{vv} : G_{tepe_tepe} , G_{ve} : $G_{tepe_ayrıtlı}$)

A ırlıklı Graf (Weighted Graph): Her ayrıtlı bir a ırlıklı (weight) veya maliyet (cost) de erinin atandı ı graftır. Örne in ekil 6.5'teki graf a ırlıklı graftır.

Yol (Path): $G(V, E)$ grafında i_1 ve i_k tepeleri arasında $P = i_1, i_2, \dots, i_k$ ekinde belirtilen tepelerin bir dizisidir (E 'de, $1 <= j < k$ olmak üzere, her j için (i_j, i_{j+1}) ekinde gösterilen bir ayrıtlı varsa). E er graf yönlü ise yolun yönü ayrıtlar ile hizalanmalıdır. ekil 6.7'deki grafa stanbul'dan Ankara'ya do rudan veya Rize'den geçerek gidilebilir.



ekil 6.7 A ırlıklı graf.

Yol için gev ek ba lı “weakly connected” denir. Herhangi bir kö eden istedi imize gidiyorsak, bu sıkı ba lıdır “strongly connected”.

Basit Yol (Simple Path): Tüm dü ümlerin farklı oldu u yoldur.

Çevre: Her bir iç tepesinin derecesi iki olan n ayrıtlı kapalı ayrıtlı katarına çevre denir ve bu da C_n ile gösterilir.

Uzunluk: Bir yol üzerindeki ayrıtların sayısı o yolun uzunlu udur.

Bir Tepenin Derecesi: Yönsüz bir grafın bir v tepesine ba lı olan ayrıtlarının sayısına v 'nin derecesi denir ve bu $degree(v)$ ile gösterilir. Self-ayrıtlar 1 olarak sayılır.

Yönlü graflarda iki çe it derece vardır. Bunlar;

Giri Dercesi (Indegree): Yönlü grafa, bir tepeye gelen ayrıtların sayısına indegree denir.

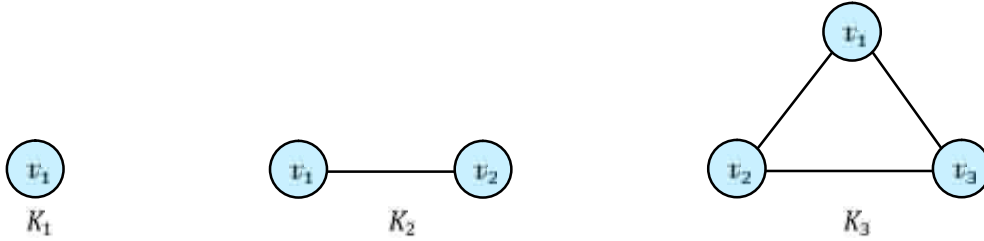
Çıkı Derecesi (Outdegree): Yönlü grafa, bir tepeden çıkan ayrıtların sayısına outdegree denilir. Örne in ekil 6.5'teki yönlü grafa C 'nin giri derecesi 2, çıkı derecesi ise 0'dır.

Ba lı Graf (Connected Graph): Her tepe çifti arasında en az bir yol olan graftır.

Alt Graf (Subgraph): H grafının tepe ve ayrıtları G grafının tepe ve ayrıtlarının alt kümesi ise H grafi G grafının alt grafıdır.

Aaç (Tree): Çevre içermeyen yönsüz bağılı graftır.

Tam Graf: Birbirinden farklı her tepe çifti arasında bir ayrıt bulunan graflardır. n tepeli bir tam graf K_n ile gösterilir ve $n \cdot (n - 1) / 2$ tane ayrıtı vardır (*kendilerine bağılantı yok*). ekil 6.8'de K_1, K_2 ve K_3 tam grafları gösterilmiştir.



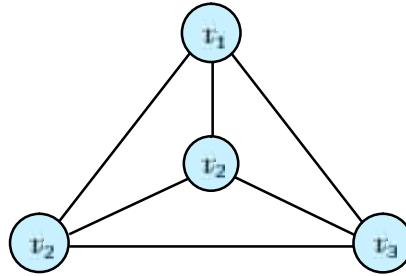
ekil 6.8 K_1, K_2 ve K_3 tam grafları.

Tam Yönlü Graf (Complete Digraph): n tepe sayısı olmak üzere $n \cdot (n - 1)$ ayrıtı olan graftır.

Seyrek Graf: Ayrıtı sayısı mümkün olandan çok çok az olan graftır.

Katlı Ayrıt: Herhangi iki tepe çifti arasında birden fazla ayrıt varsa, böyle ayrıtlara *katlı ayrıt* denir.

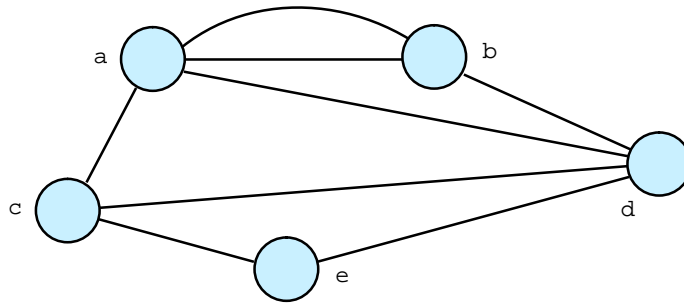
Tekerlek (Wheel) Graf: Bir cycle grafına ek bir tepe eklenerek oluşturulan ve eklenen yeni tepenin, diğer bütün tepelere bağılı olduğu graftır. W_n ile gösterilir.



ekil 6.9 Tekerlek graf.

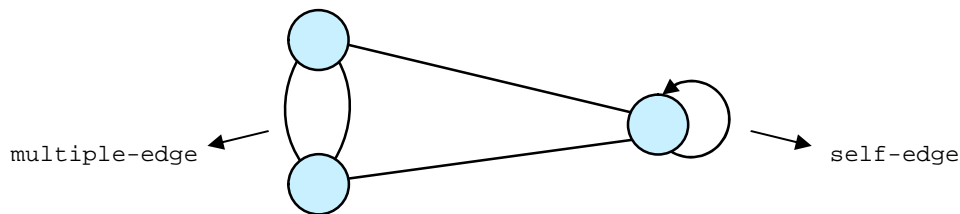
Daire veya Devir (Cycle): Bağılantı ve bitiş tepeleri aynı olan basit yoldur. ekil 6.7'deki İstanbul-Rize-Ankara-İstanbul örneği.

Paralel Ayrıtlar (Parallel Edges): iki veya daha fazla ayrıt bir tepe çifti ile bağılantıdır. ekil 6.10'da a ve b iki paralel ayrıt ile birleştirilmiştir.



ekil 6.10 Paralel ayrıtlı bir graf.

Çoklu (Multi) Graf: Multigraf iki tepe arasında birden fazla ayrıta sahip olan veya bir tepenin kendi kendisini gösteren ayrıta sahip olan graftır.



ekil 6.11 Çoklu (*multi*) graf.

Spanning Tree: G 'nin tüm tepelerini içeren bir ağaç şeklindeki alt graflardan her birine *spanning tree* denir.

Forest: Bağılı olmayan ağaç topluluğudur.

6.2 GRAFLARIN BELLEK ÜZERİNDE TUTULMASI

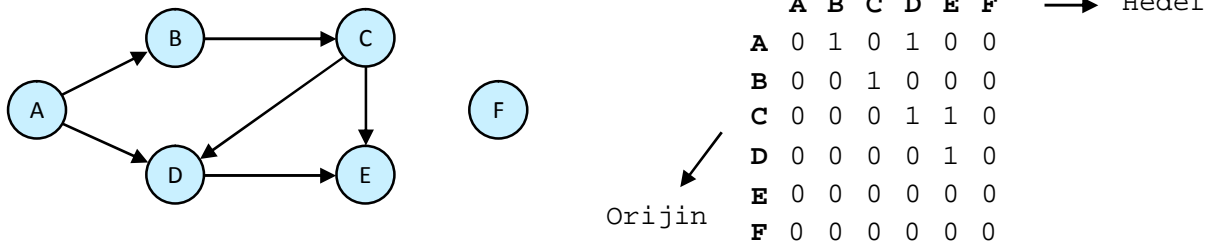
Grafların bellekte tutulması veya gösterilmesi ile ilgili birçok yöntem vardır. Bunlardan birisi, belki de en yalın olanı, doğrudan matris şeklinde tutmaktır; grafin komşuluk matrisi elde edilmişse, bu matris doğrudan programlama dilinin matris bildirim özellikleri uyarınca bildirilip kullanılabilir. Ancak grafin durumu ve uygulamanın gereksinimine göre diğer yöntemler daha iyi sonuç verebilir. Bir $G = (V, E)$ grafinin bilgisayara aktarılmasındaki en yaygın iki standart uygulamaları şunlardır:

- Komşuluk matrisi ile ardışık gösterim (*adjacency matrix representation*),
- Başlı listeler ile başlı gösterim veya komşuluk yapısı ile gösterim (*adjacency list representation*).

Bu uygulamalar hem yönlü graflar hem de yönsüz grafların her ikisinde de geçerlidir. Genelde, yoğun graflar ($|E|$ 'nin $|V^2|$ 'ye yakın olduğu graflar) için ya da iki tepeyi birbirine bağlayan ayrıntı varlığını hızlı bir şekilde belirlemek için matrisler kullanılır. Seyrek graflar ($|E|$ 'nin $|V^2|$ 'den çok daha az olduğu graflar) için ise başlı listeler kullanılır.

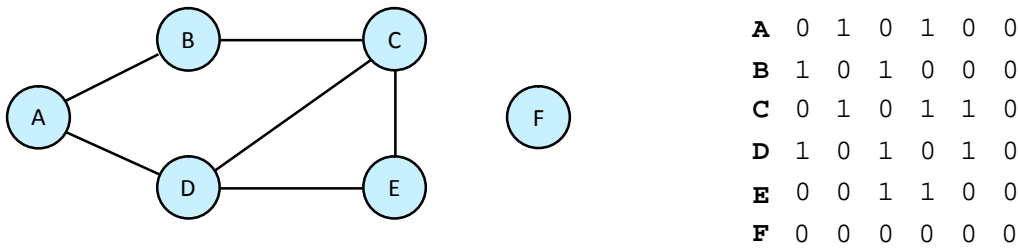
Komşuluk Matrisi (Adjacency Matrix)

Tepelerden tepelere olan bağlantıyı boolean değerlerle ifade eden bir kare matristir. n boyutlu bir dizi ile implemente edilir. $G(V, E)$, $|v|$ düğümlü bir graf ise $|v| \times |v|$ boyutunda bir matris ile bir G grafi bellek ortamında gerçekleştirilebilir. Eğer $(A, B) \in E$ ise 1, diğer durumlarda ise 0 olarak işaretlenir. Yönsüz bir grifta matris simetrik olacaktır dolayısıyla bellek tasarrufu amacıyla alt yarı üçgen ya da üst yarı üçgen kullanmak yerinde olur. Her iki durumda da gerçekleştirimin bellek karmaşıklığı $O(v^2)$ olur.



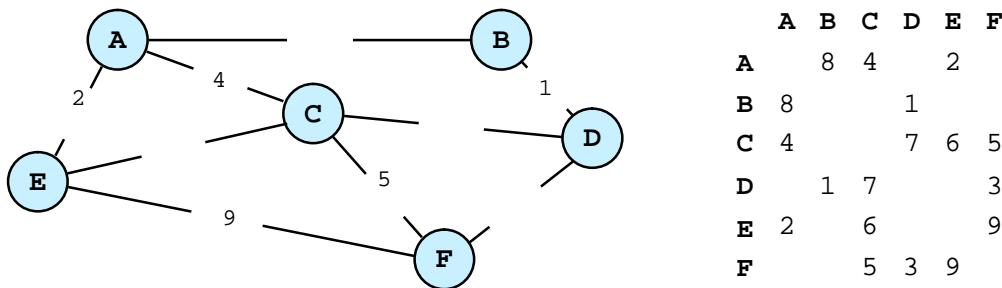
ekil 6.12 Yönlü bir grafin komşuluk matrisi ile gösterimi.

Yönlü grafların matrisi genelde simetrik olmaz fakat yönsüz graflarda bir simetriklik söz konusudur. İndi ekil 6.13'te yönsüz bir grafin komşuluk matrisiyle gösterimi yer almaktadır.



ekil 6.13 Yönsüz bir grafin komşuluk matrisi ile gösterimi.

Aynı ayrıntı bilgilerinin, örneğin (A, B) ve (B, A) ayrıntılarının her ikisinin birden depolanması biraz gereksiz gibi görünebilir fakat ne yazık ki bunun kolay bir yolu yoktur. Çünkü ayrıntılar yönsüzdür ve parent ve child kavramları da yoktur. ekil 6.14'te ağırlıklı ancak yönlendirilmemiş bir grafin komşuluk matrisi gösterilmiştir.



ekil 6.14 Yönlendirilmemiş ağırlıklı bir grafin komşuluk matrisi ile gösterimi.

Yukarıdaki ekilerde komşuluk matrisleri ile gösterilen grafların bilgisayardaki uygulamasını `int a[6][6];` şeklinde iki boyutlu bir dizi şeklinde gerçekleştirebiliriz.

Örnek 6.1: Komuluk matrisi ile temsil edilen yönlü bir grafta bir düümün giri ve çıkı derecelerini bulan fonksiyonu yazalım. Grafta 6 adet tepe oldu u varsayalım. Bir düümün derecesini bulmak için ilgili satır ya da sütundaki 1'ler sayılmalıdır ($O(n)$). Satırdaki 1'ler outdegree, sütundakiler ise indegree de erini verir. Bu ikisinin toplamı ise o düümün derecesidir ($O(2n)$).

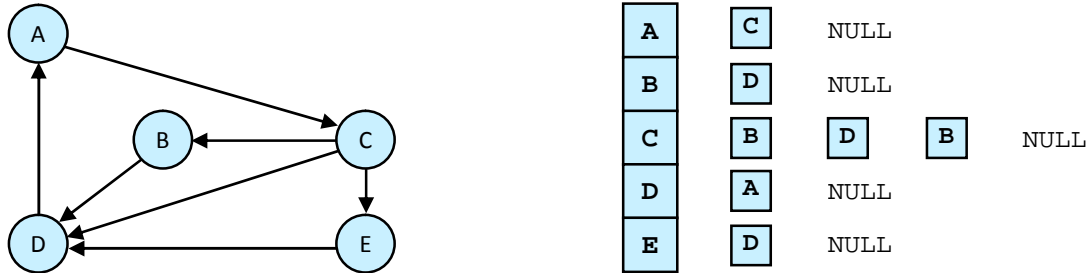
```
#define tepe_sayisi 6 // tepe sayısının 6 oldu u varsayılıyor
/* Yönlü bir grafta giriş derecesini veren fonksiyon */
int indegree(int a[][tepe_sayisi], int v) {
    int i, degree = 0;
    for(i = 0; i < tepe_sayisi; i++)
        degree += a[i][v]; // v sütunundaki 1'ler toplanıyor
    return degree; // giriş derecesi (indegree) geri döndürülüyor
}
```

Çıkı derecesini bulan fonksiyon için yalnızca $a[i][v]$ yerine $a[v][i]$ yazmak yeterlidir.

```
/* Yönlü bir grafta çıkış derecesini veren fonksiyon */
int outdegree(int a[][tepe_sayisi], int v) {
    int i, degree = 0;
    for(i = 0; i < tepe_sayisi; i++)
        degree += a[v][i]; // v satırındaki 1'ler toplanıyor
    return degree; // çıkış derecesi (outdegree) geri döndürülüyor
}
```

Komuluk Listesi (Adjacency List)

Her v tepesi kendinden çıkan ayrıtları gösteren bir ba lı listeye sahiptir ve her bir tepe için kom u tepelerin listesi tutulur. Bellek gereksinimi $O(|V| + |E|)$ dizi boyutu + ayrıt sayısı eklindedir. Komuluk listesi seyrek graflar için hem zaman hem de bellek açısından daha verimlidir fakat tam bir graf veya yo un graflar için verim daha azdır.



ekil 6.15 Yönlü bir grafın komuluk listesi ile gösterimi.

Komuluk listesi için veri yapısı a a ıdaki gibi tanımlanabilir;

```
#define tepe_sayisi 6 // tepe sayısının 6 oldu u varsayılıyor
struct vertex {
    int node;
    struct vertex *nextVertex;
};
struct vertex *head[tepe_sayisi]; // 6 boyutlu, bir ba lı liste başı
```

Komuluk Matrisleri ve Komuluk Listelerinin Avantajları-Dezavantajları

Komuluk matrisi

- Çok fazla alana ihtiyaç duyar. Daha az hafıza gereksinimi için seyrek matris teknikleri kullanılmalıdır.
- Herhangi iki düümün komuluk olup olmadı na çok kısa sürede karar verilebilir.

Komuluk listesi

- Bir düümün tüm komuluklarına hızlı bir ekilde ulaşılır.
- Daha az alana ihtiyaç duyar.
- Olu turulması matrise göre daha zor olabilir.

Kaynaklar : Veri Yapıları ve Algoritmalar, *Rıfat Çölkesen*
C ile Veri Yapıları, *Prof. Dr. brahim Akman*

E-mail : hakankutucu@karabuk.edu.tr

Web : <http://web.karabuk.edu.tr/hakankutucu/BLM227notes.htm>