

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

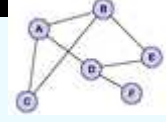


çindekiler

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
ÖZYİNELEMELE 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 DOĞRUSAL LİSTELER	19
Tek Bağlı Doğrusal Liste Oluşturmak ve Eleman Ekleme	19
Tek Bağlı Doğrusal Listenin Başına Eleman Ekleme	20
Tek Bağlı Doğrusal Listenin Sonuna Eleman Ekleme	20
Tek Bağlı Doğrusal Liste Elemanlarının Tüm Bilgilerini Yazdırmak	20
Tek Bağlı Doğrusal Listenin Elemanlarını Yazdırmak	20
Tek Bağlı Doğrusal Listenin Elemanlarını Saymak	21
Tek Bağlı Doğrusal Listelerde Arama Yapmak	21
Tek Bağlı Doğrusal Listelerde Ki Listeyi Birleştirmek	22
Tek Bağlı Doğrusal Listelerde Verilen Bir Değere Sahip Düğümü Silmek	22
Tek Bağlı Doğrusal Listelerde Verileri Tersten Yazdırmak	22
Tek Bağlı Doğrusal Listenin Kopyasını Oluşturmak	23
Tek Bağlı Doğrusal Listeyi Silmek	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 Değere Sahip Düğümü Silmek	27
ÇİFT BAĞLI DOĞRUSAL (<i>Double Linked</i>) LİSTELER	28
Çift Bağlı Doğrusal Listenin Başına Eleman Ekleme	28
Çift Bağlı Doğrusal Listenin Sonuna Eleman Ekleme	28
Çift Bağlı Doğrusal Listelerde Araya Eleman Ekleme	29
Çift Bağlı Doğrusal Listelerde Verilen Bir Değere Sahip Düğümü Silmek	29
Çift Bağlı Doğrusal Listelerde Arama Yapmak	30
Çift Bağlı Doğrusal Listelerde Karşılaştırmayı 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 (push)	34
Bir Stack'in Tüm Elemanlarını Silme lemi (reset)	34
Stack'lerden Eleman Çıkarma lemi (pop)	34
STACK'LER N BA LI L STE (<i>Linked List</i>) MPLEMENTASYONU	35
Stack'in Bo Olup Olmadı ının Kontrolü (isEmpty)	36
Stack'in Dolu Olup Olmadı ının Kontrolü (isFull)	36
Stack'lere Yeni Bir Dü üm Ekleme (push)	36
Stack'lerden Bir Dü ümü Silme (pop)	37
Stack'in En Üstteki Verisini Bulmak (top)	37
Bir Stack'e Ba langıç De erlerini Verme (initialize)	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 (initialize)	43
Kuyru un Bo Olup Olmadı ının Kontrolü (isEmpty)	43
Kuyru un Dolu Olup Olmadı ının Kontrolü (isFull)	43
Kuyru a Eleman Ekleme (enqueue)	43
Kuyruktan Eleman Çıkarma lemi (dequeue)	44
KUYRUKLARIN BA LI L STE (<i>Linked List</i>) MPLEMENTASYONU	44
Kuyru a Ba langıç De erlerini Verme (initialize)	44
Kuyru un Bo Olup Olmadı ının Kontrolü (isEmpty)	44
Kuyru un Dolu Olup Olmadı ının Kontrolü (isFull)	45
Kuyru a Eleman Ekleme (enqueue)	45
Kuyruktan Eleman Çıkarma lemi (dequeue)	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 aca Veri Ekleme	56
K L ARAMA A AÇLARI (BSTs - <i>Binary Search Trees</i>)	59
kili Arama A acına Veri Ekleme	59
Bir A acın Dü ümlerinin Sayısını Bulmak	59
Bir A acın Yüksekli ini Bulmak	60
kili Arama A acından Bir Dü üm Silmek	61
kili Arama A acında Bir Dü ümü Bulmak	64
kili Arama A acı Kontrolü	65
kili Arama A acında Minimum Elemanı Bulmak	65
kili Arama A acında Maximum Elemanı Bulmak	65
Verilen ki A acı Kar ıla tırmak	65
Alı tırmalar	65
AVL A AÇLARI	66
Önerme	68
spat	68
Bir AVL A acının Yapısı	70
spat	70
ddia	71
spat	71
AVL A açlarında Ekleme lemi	72
Bir AVL A acında Dü ümleri Döndürmek	73
Tek Döndürme (<i>Single Rotation</i>)	73
Çift Döndürme (<i>Double Rotation</i>)	76
AVL A açlarında Silme lemi	79
ÖNCEL KL KUYRUKLAR (<i>Priority Queues</i>)	81
Binary Heap (<i>kili Yı ın</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

1.1 GİRİŞ

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, iaretçiler kullanılarak daha etkin şekilde hafızada saklanabilir. Veri yapıları, dizi ve iaretçiler ile yapılmasının yanında, nesneler ile de gerçekleştirilebilir.

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, "Ya mur" ve, "Merhaba" stringleri, 0,1 mantıksal değerleri, ses ve resim sinyalleri vb.) tanımlarıyla 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, "Ya mur" öğrencinin ismi, "Merhaba" selamlama kelimesi stringleri, boş anlamında 0, dolu anlamında 1 mantıksal değerleri, anlamı bilinen ses ve resim sinyalleri verilerin bilgi haline dönüşüm halleridir.

Veriler büyüklüklerine göre bilgisayar belleğinde farklı boyutlarda yer kaplarlar. Büyüklüklerine, kapladıkları alan boyutlarına ve tanım aralıklarına göre veriler Veri Tip'leri ile sınıflandırılırlar. 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	$1.7 \times 10^{-308} - 1.7 \times 10^{308}$

* İlemciye göre 16 veya 32 bitlik olabilmektedir.

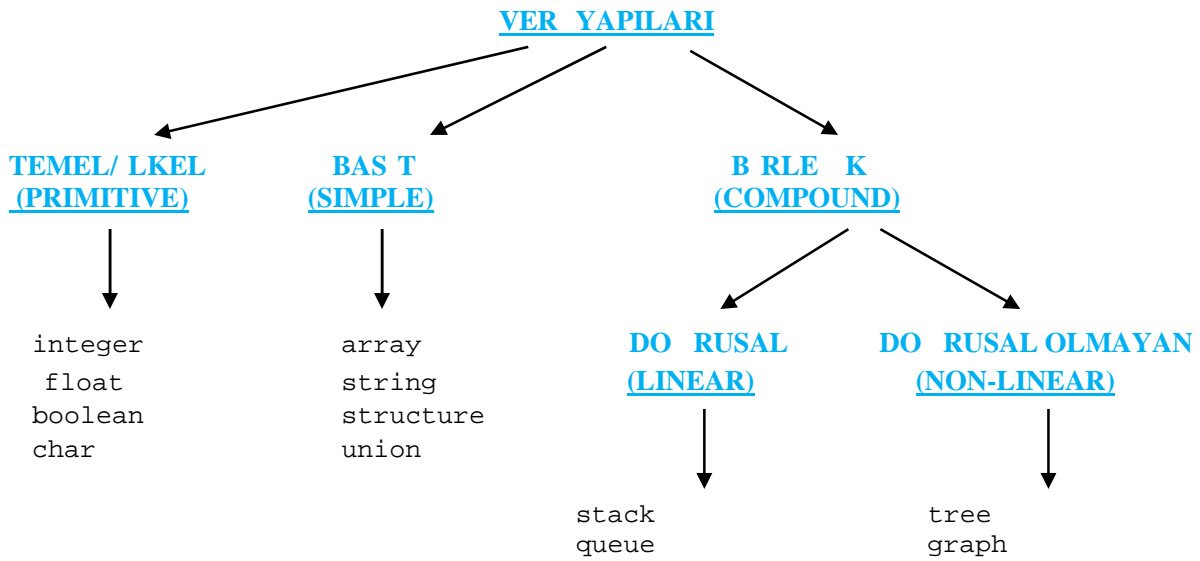
Tablo1.1 C'de veri tipleri ve tanım aralıkları.

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 aşamasıdır. 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/ İlkel (*primitive*), Basit (*simple*), Birlik (*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 birlikler (*unions*)'dır.
- Birlik 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 en etkin biçimde faydalanma olanakları sunar, ayrıca programın hızını ve etkinliğini artırır, maliyetini düşürür.

Veriden Bilgiye Geçiş

Veriler bilgisayar belleğinde 1 ve 0'lardan 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;

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 gerçe kler 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 gerçe kler ir. letim sistemi dizinin her verisini (*elemanını*) 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 gerçe kler ir. letim sistemi stringin her verisini (*elemanını*) 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. öğrenci, kayit yapısından bir de i kendir ve öğrenci de i keni üyelerine a a ıdaki veri atamalarını yaptıktan sonra, bellekte ekil 1.3 d)'deki gibi bir yerle im gerçe kler 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 içeriisindeki 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

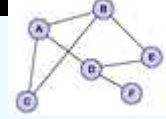
    printf("%p\n", agirlik); // agirlik dizisinin adresini yaz, 0022FF20 yazılır
    printf("%p\n", p); // p de işkeninin verisini yaz, 0022FF20 yazılır
    printf("%d\n", agirlik[0]); // Dizinin ilk elemanının verisini yaz, 48 yazılır
    printf("%d\n", *p); // p de işkeninde bulunan adresteki veriyi yaz, 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 başlılı doğrusal listeler,
 - b. Bir başlılı dairesel listeler,
 - c. İkibaşlılı doğrusal listeler,
 - d. İkibaşlılı dairesel listeler,
- 2) Listeler ile stack (yığın) uygulaması,
- 3) Listeler ile queue (kuyruk) uygulaması,
- 4) Listeler ile dosyalama uygulaması,
- 5) Çok başlılı listeler,
- 6) Ağaçlar,
- 7) Matrisler,
- 8) Arama algoritmaları,
- 9) Sıralama algoritmaları,
- 10) Graflar.



BÖLÜM

Veri Yapıları 2

2.1 G R

Büyük bilgisayar programları yazarken karşılaştığımız en büyük sorun programın hedeflerini tayin etmek değildir. 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 baskılar ve ardından yapılan işin programa dökülmesinin çok kolay olduğunu fark eder. Lakin söz konusu bilgilerin geliştirilirmekte olan programın baskı 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 baskılar. Zayıf bir şekilde birbirine baskılan program özellikleri bu aşamada da ılıp işi göremez hale kolaylıkla gelebilir.

Bilgisayar ile ilgili işlerde en baskıta 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 karşılaştırmaya, arama, yeni veri ekleme, silme gibi işlemleri verebiliriz ki bunlar ayrılmış olarak bellekte gerçekleştirilir. 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 baskı 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 baskıında 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 baskılangıcından sonuna kadar tutulmak 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 edilmiştir ve baskı 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 çağırıldığında `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 anlaşılabilceği 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 yazılmış 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 ını ve sonucu da string olarak tanımlanmı "sonuc" de i kenine attı ını 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 dönu tipti yok iken, ikinci programda ise char* olarak dönu tipti 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 çalıştırılmasından 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 çağırısında argümanlar de ere göre çağırma ile geçirilirse, argümanın de erinin bir kopyası oluşturulur ve çağırılan fonksiyona geçirilir. Oluşturulan kopyadaki de i iklikler, çağırıcıdaki orijinal de i kenin de erini etkilemez. Bir argüman referansa göre çağırıldı ında ise çağırıcı, çağırılan fonksiyonun de i kenin orijinal de erini ayarlamasına izin verir. ki örnekle konuyu hatırlatalım.

Örnek 2.1 Swap işlemi için call by value ve call by reference yöntemiyle fonksiyonlara çağırısı yapılıyor.

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

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

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

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

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

    getch();
    return 0;
}
```

2.2 ÖZY NELEMEL FONKS YONLAR

Özyinelemeli (*rekürsif*) fonksiyonlar kendi kendini çağırılan 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 çözme 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 çıkışı (*exitcase*) durumu da denir.
2. Yukarıdaki tanıımı da içeren problemi aynı formda küçük alt problemlere parçalayan recursive case'dir.

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

Rekürsif bir fonksiyonun genel yapısı

Her rekürsif fonksiyon mutlaka *if* segmenti içermelidir. E er içermezse rekürsif durumla base durumu ayırt edilemez. Bu segmentin içerisinde de bir base-case artı olmalıdır. E er base-case durumu sağlanıyorsa sonuç rekürsif bir uygulama olmaksızın iteratif (*özyinelemesiz*) olarak hesaplanır. E er 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,

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

Bu da bir temel durum içerir. Bu temel durum, e er $n = 0$ ise sonuç 1'dir. De ilse $n \cdot (n-1)!$ olacaktır. İmdi bunu koda dökelim;

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

Fonksiyona 4 rakamını gönderdi imizi düşünelim. 4 sıfıra e it olmadı ından fonksiyon else kısmına gidecek ve $4 * \text{fact}(3)$ ekleyle kendini 3 rakamıyla tekrar çağıracaktır. Bu adımlar aşağıda gösterilmiştir;

<pre>4 * fact(3) ↓ 3 * fact(2) ↓ 2 * fact(1) ↓ 1 * fact(0) ↓ (1)</pre>	<p>ekilde görüldü ü gibi en alttaki fonksiyon çağırısında iteratif durum gerçekleşti için fonksiyon sonlanacak ve ilk çağırıldı ı noktaya geri dönecektir. Bu esnada bellekte $\text{fact}(0)$ çağırısı yerine 1 yazılacağı için o satırdaki durum $1 * 1$ eklini alacak ve 1 sonucu üretilecektir. Bu sefer $\text{fact}(1)$ çağırısının yapıldığı yere dönecek ve yine $2 * 1$ eklini alıp 2 üretecektir. Sonra $\text{fact}(3)$ çağırısının olduğu satırda $3 * 2$ eklini 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 de i ikliklerle 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 nesneleri 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 eilde 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 görece 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 n 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 (*modifiye*),
- 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 arttırılması gerekir,
- Silme işlemlerinde de diğer bütün elemanlar ötelenmelidir.

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'a hem de önceki node'a bağlantı vardır. Bu bağlantılar Forward Link (*ileri başlı*) ve Backward Link (*geri başlı*) olarak adlandırılırlar. Doğrusal listelerde listede bulunan en son node'un başına hiçbir node'a bağlantısı yoktur. Başına deeri 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. İlgilikli bilinmesi gereken ayrıntı, çift başlı listelerde `previous` adında ikinci bir pointer daha vardır. Diğerleri aynı yapıdadır.

Bağlı Listeler ile İşlemler

Bağlı listeler üzerinde;

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

vb. gibi ve kuşkusuz daha fazla işlemler yapılabilir. Bu işlemlerden bazılarını açıklayalım ve fonksiyon halinde yazalım.

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>