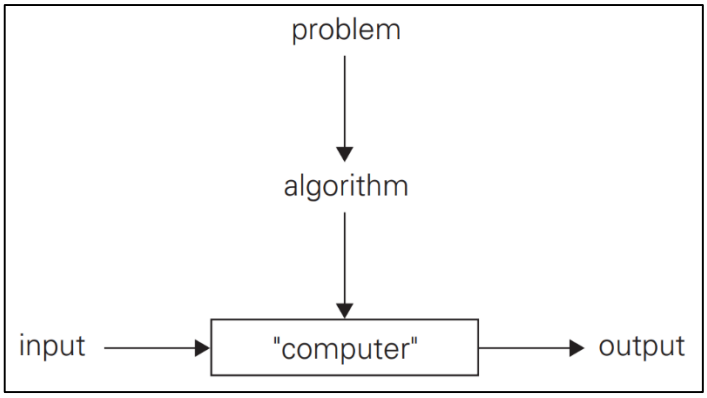
3 Algoritma Analizi

3.1 Algoritma Nedir?

Bu kavramı açıklamak için evrensel olarak üzerinde anlaşmaya varılmış bir ifade olmamasına rağmen, kavramın ne anlama geldiği konusunda genel bir anlaşma vardır. Algoritma, bir sorunu çözmek için, başka bir deyişle bir sistemde herhangi uygun bir girdi için sınırlı bir süre içerisinde gerekli bir çıktıyı elde etmeyi amaçlayan açık bir talimatlar dizisidir.

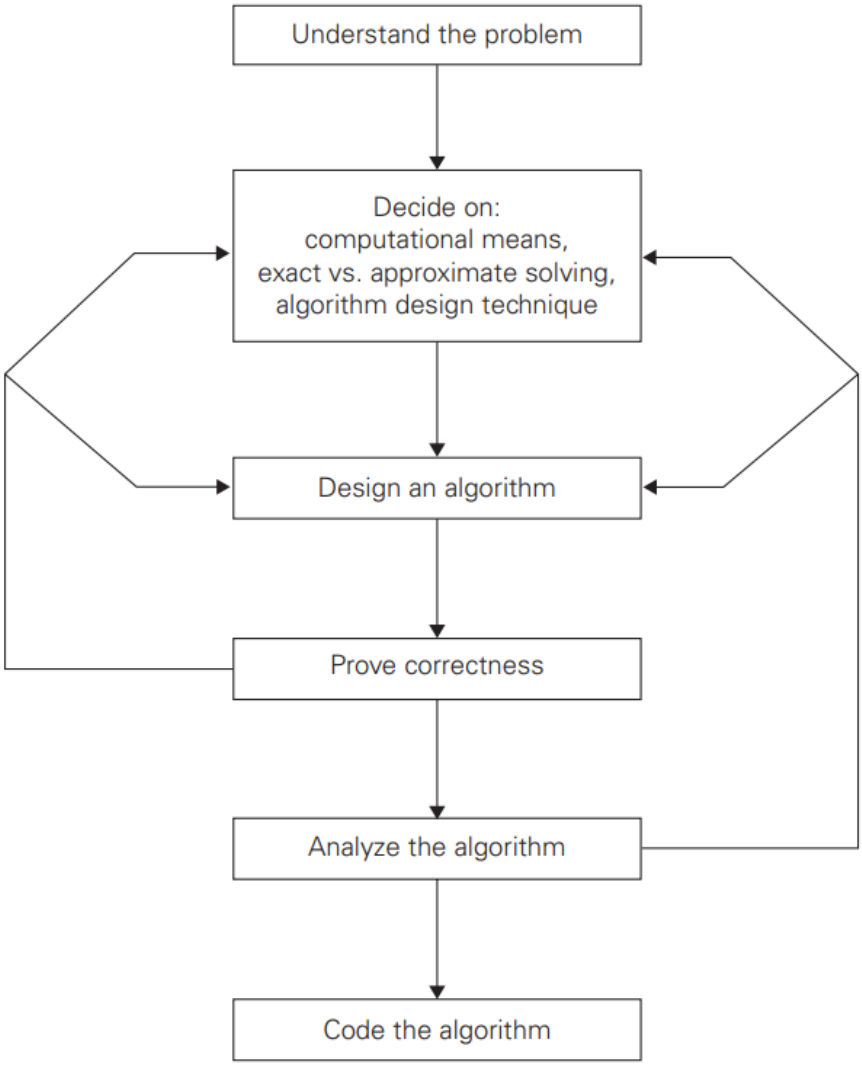


Şekil 1. Algoritma Kavramı

3.2 Algoritmik Problem Çözmenin Temelleri

Algoritmaları problemlere getirilen prosedürel çözümler olarak düşünebiliriz. Bu çözümler cevaplar değil, cevapları almak için özel talimatlardır. Bilgisayar bilimini diğer disiplinlerden farklı kılan tam olarak tanımlanmış prosedürlerdir.

Bir algoritmanın tasarlanması ve analizinde genellikle bir dizi adımdan faydalanılır. Bu adımları Şekil 2'de görebiliriz.



Şekil 2. Algoritma tasarım ve analiz süreci

4 Veri Tipleri

4.1 Giriş

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 içerisinden seçilebileceği gibi, işaretçiler kullanılarak daha etkin şekilde hafızada saklanabilir. Veri yapıları, dizi ve işaretçiler 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, "Yağmur" ve, "Merhaba" stringleri, 0,1 mantıksal değerleri, ses ve resim sinyalleri vb.) tanımıyla ifade edilebilir.

Bilgi ise, verinin işlenmiş 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 şubesi 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 dönüşmüş halleridir.

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

Tablo. Veri Tipleri ve Tanım Aralıkları

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 shortint	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 x 10-38 - 3.4 x 1038
double	64	1.7 x 10-308 – 1.7 x 10308

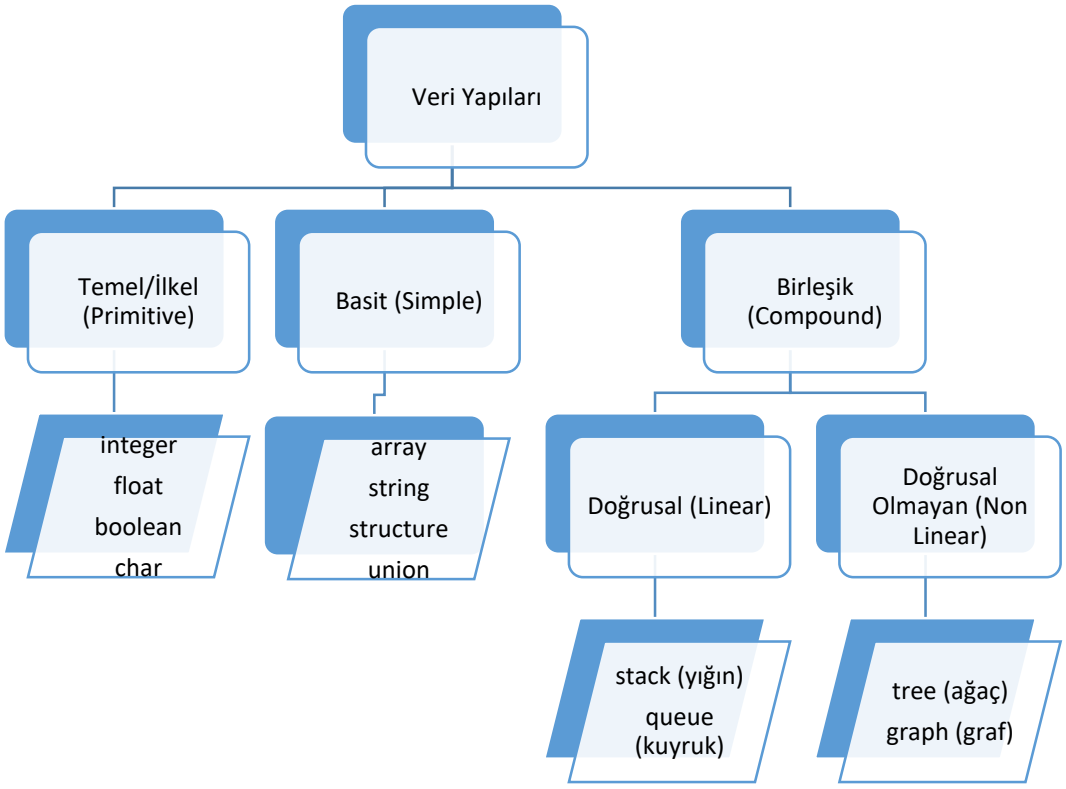
*İşlemciye göre 16 veya 32 bitlik olabilmektedir.

Her programlama dilinin tablodakine benzer, kabul edilmiş veri tipi tanımlamaları vardır. Programcı, programını yazacağı problemi incelerken, program algoritmasını oluştururken, programda kullanacağı 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 yaptırtır. 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.

4.2 Veri Yapısı

Verileri tanımlayan veri tiplerinin, birbirleriyle ve hafızayla ilgili tüm teknik ve algoritmik özellikleridir. C dilinin Veri Yapıları aşağıdaki şekildeki gibi sınıflandırılabilir.



Şekilden de görüleceği üzere, C Veri Yapıları Temel/İlkel (primitive), Basit (simple), Birleşik (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 birleşimler (unions)'dir.
- Birleşik 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 arttırır, maliyetini düşürür.

4.3 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;

0100 0010	0100 0001	0100 0010	0100 0001
B	A	B	A

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

0100	0010	0100	0001	0100	0010	0100	0001
4	2	4	1	4	2	4	1

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

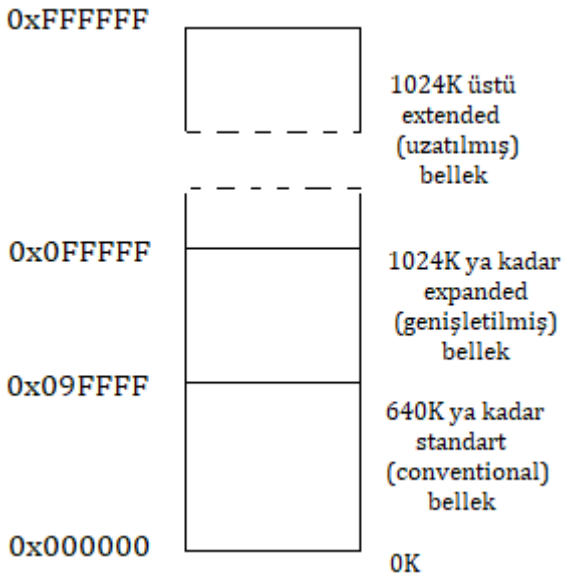
0100 0010 0100 0001	0100 0010 0100 0001
16961	16961

Bu veri işaretsiz 32 bitlik tamsayı ise, 32 bitlik bütün grup olarak bir işaretsiz tamsayıya karşılık düşer;

0100 0010 0100 0001 0100 0010 0100 0001
1111573057

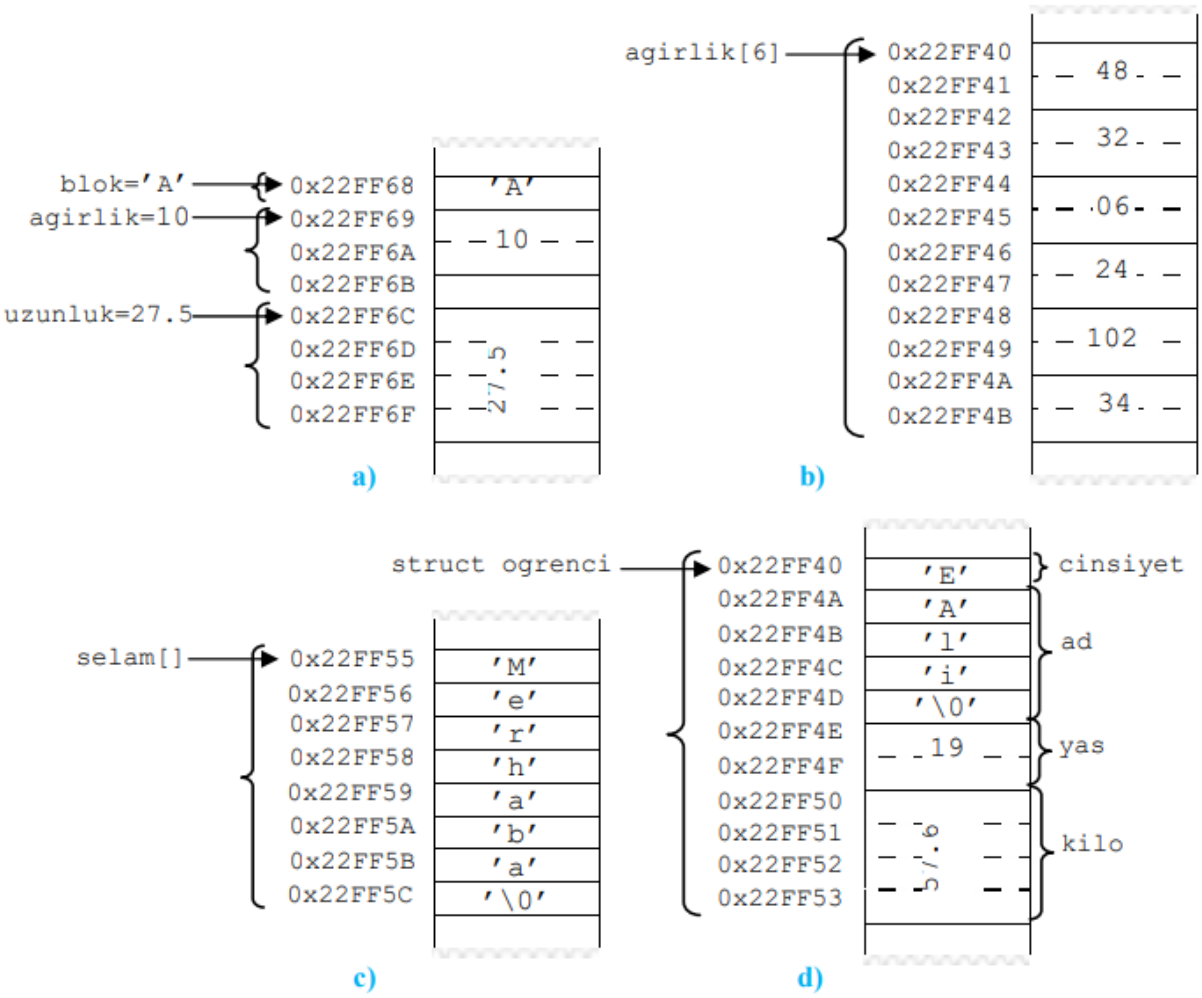
4.4 Belleğin Yapısı ve Veri Yapıları

Yapısal olarak bellek, büyüklüğüne bağlı olarak binlerce, milyonlarca 1'er Byte (8 Bit)'lik veriler saklayabilecek biçimde tasarlanmış bir elektronik devredir. Bellek, aşağıdaki şekildeki gibi her byte'ı bir hücre ile gösterilebilecek büyükçe bir tablo olarak çizilebilir.



Şekil. Bellek haritası

Her bir hücreyi birbirinden ayırmak için hücreler numaralarla adreslenir. Program, işlemci ve işletim sistemi bu adresleri kullanarak verilere erişir (yazar/okur). Büyük olan bu adres numaraları 0'dan başlayarak bellek boyutu kadar sürer ve 16'lık (Hexadecimal) biçimde ifade edilir. İşletim sistemleri belleğin bazı bölümlerini kendi dosya ve programlarını çalıştırmak için, bazı bölümlerini de donanımsal gereksinimler için ayırır ve kullandırır. Ancak belleğin büyük bölümü uygulama programlarının kullanımına ayrılmıştır. Aşağıdaki şekilde DOS işletim sistemi için bellek haritası görülmektedir.



Şekil. Veri yapılarının bellek yerleşimleri

Temel/İlkel (Primitive) veri yapılarından birisinin tipi ile tanımlanan bir değişken, tanımlanan tipin özelliklerine göre bellekte yerleştirilir. Örneğin;

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

değişken tanımlamaları sonunda bellekte yukarıdaki şekil-a'daki gibi bir yerleşim gerçekleşir. İşletim sistemi, değişkenleri bellekteki veri tiplerinin özelliklerine uygun boş alanlara yerleştirir.

Basit (Simple) veri yapıları temel veri yapıları ile oluşturulur. Örneğin;

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

tanımlamasındaki `agirlik` dizisi 6 adet `int` (tamsayı) veri içeren bir veri yapısıdır. Bellekte şekil-b'deki gibi bir yerleşim gerçekleşir. İşletim sistemi dizinin her verisini (elemanını) ardı ardına, bellekteki boş alanlara veri tipinin özelliklerine uygun alanlarda yerleştirir. Örneğin;

```
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 şekil-c'deki gibi bir yerleşim gerçekleşir. İşletim sistemi stringin her verisini (*elemanını*) ardı ardına, bellekteki boş alanlara veri tipinin özelliklerine uygun alanlarda yerleştirir. Örneğin;

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

tanımlamasında `kayit` adında bir `structure` (*yapı*) oluşturulmuştur. Bu veri yapısı dikkat edilirse farklı temel veri yapılarından oluşan, birden çok değişken tanımlaması (*üye*) içermektedir. `ogrenci`, `kayit` yapısından bir değişkendir ve `ogrenci` değişkeni üyelerine aşağıdaki veri atamalarını yaptıktan sonra, bellekte şekil-d'deki gibi bir yerleşim gerçekleşir.

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

İşletim sistemi `kayit` veri yapısına sahip öğrenci değişkeninin her bir üyesini ardı ardına ve bir bütün olarak bellekteki boş alanlara üyelerin veri tiplerinin özelliklerine uygun alanlarda yerleştirir.

Birleşik (Compound) veri yapıları basit veri yapılarından dizi veya structure tanımlamaları ile oluşturulabileceği gibi, nesne yönelimli programlamanın veri yapılarından class (sınıf) tanımlaması ile de oluşturulabilir. İlerleyen bölümlerde `structure` ile yeni veri yapılarının tanımlama ve uygulamaları anlatılmaktadır.

4.5 Adres Operatörü ve Pointer Kullanımı

Daha önce yaptığımız veri yapıları tanımlamalarında, verilere değişken adları ile erişilebileceği görülmektedir. Aynı verilere, değişkenlerinin adresleriyle de erişilebilir. Bu erişim tekniği bazen tercih edilebilir olsa da, bazen kullanılmak zorunda kalınabilir.

Aşağıdaki kodlar ile temel veri yapılarının adreslerinin kullanımları incelenmektedir. `printf()` fonksiyonu içerisindeki formatlama karakterlerine dikkat ediniz. Adres değerleri sistemden sisteme farklılık gösterebilir.

```
main() {
     int agirlik = 10;
     int *p;
     p = &agirlik;
     printf("%d\n", agirlik); // agirlik değişkeninin verisini yaz, 10 yazılır
     printf("%p\n", &agirlik); // agirlik değişkeninin adresini yaz, 0022FF44 yazılır
     printf("%p\n", p); // p değişkeninin verisini yaz, 0022FF44 yazılır
     printf("%d\n", *p); // p değişkenindeki adresteki veriyi yaz, 10 yazılır
     printf("%p\n", &p); // p değişkeninin 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 değişkeninin verisini yaz, 0022FF20 yazılır
     printf("%d\n", agirlik[0]); // Dizinin ilk elemanının verisini yaz, 48 yazılır
     printf("%d\n", *p); // p değişkeninde bulunan adresteki veriyi yaz, 48 yazılır
     printf("%d\n", agirlik[1]); // Dizinin ikinci elemanının verisini yaz, 32 yazılır
     printf("%d\n", *++p); // p değişkenindeki 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.

4.6 Yaygın Olarak Kullanılan Veri Yapıları Algoritmaları

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

5 Veri Yapıları

Büyük bilgisayar programları yazarken karşılaştığımız en büyük sorun programın hedeflerini tayin etmek değildir. Hatta programı geliştirme aşamasında hangi metotları kullanacağımız hususu da bir problem değildir. Örneğin bir kurumun müdürü "tüm demirbaşlarımızı tutacak, muhasebemizi yapacak kişisel bilgilere erişim sağlayacak ve bunlarla ilgili düzenlemeleri yapabilecek bir programımız olsun" diyebilir. Programları yazan programcı bu işlemlerin pratikte nasıl yapıldığını tespit ederek yine benzer bir yaklaşımla programlamaya geçebilir. Fakat bu yaklaşım çoğu zaman başarısız sonuçlara gebedir. Programcı işi yapan şahıstan 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 karmaşıklaşır. Biraz şans, biraz da programcının kişisel mahareti ile sonuçta ortaya çalışan 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 öğeleri bu aşamada dağılıp iş 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 ağırlıklı olarak bellekte gerçekleştirilirler. Bu işlemlerin direk 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 de olabilir. Buna **pointer** (*işaretçi*) 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;
}
```

şeklinde tanımladığımız iki veri için C derleyicisi programın başlangıcından 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 değerlerinde yapılacak olan değişiklikleri kaydederek içinde tutar. Bu bellek yerleri program boyunca statik'tir. 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 işaretçi değişkenleri için new fonksiyonu çağrıldığı 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 anlaşılabileceği üzere bu değişkenler için başlangıçta herhangi bir bellek yeri ayrılması söz konusu değildir. Bu komutlar bir döngü içerisine yerleştirildiği zaman her seferinde söz konusu alan kadar bir bellek alanını heap'den alırlar. Dolayısıyla bir programın heap alanından başlangıçta ne kadar bellek isteminde bulunacağı belli değildir. Dolayısı ile programın yürütülmesi esnasında bellekten yer alınması ve geri iade edilmesi söz konusu olduğundan buradaki işlemler dinamik yer ayrılması olarak adlandırılır. Kullanılması sona eren bellek yerleri ise:

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

komutu ile iade edilir. Söz konusu değişkenler ile işimiz bittiği 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 yaptığımızı farz edelim ve stoktaki ürünler hakkında bazı bilgileri (*kategorisi, ürün adı, miktarı, birim fiyatı* vs.) girelim. Bu veriler tür itibarı ile tek bir değişken ile tanımlanamazlar yani bunları sadece bir tamsayı veya real değişken ile tanımlayamayız çünkü bu tür veriler genelde birkaç türden değişken içeren karmaşık yapı

tanımlamalarını gerektirirler. Dolayısıyla biz söz konusu farklı değişken türlerini içeren bir **struct** yapısı tanımlarız.

Değişik derleyiciler değişik verilerin temsili için bellekte farklı boyutlarda yer ayırma yolunu seçerler. Örneğin bir derleyici bir tamsayının tutulması için bellekten 2 byte'lık bir alan ayırırken, diğer 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 bağlı olarak bellekten ilgili değişkenlerin boyutlarına uygun büyüklükte bir bloğu ayırma yoluna gider. Dolayısıyla stoktaki ürüne ait her bir veri girişinde bellekten bir blokluk yer isteniyor demektir. Böylece bellek, nerede boşluk varsa oradan 1 blokluk yer ayırmaktadır. Hem hızı arttırmak hem de işi kolaylaştırmak için her bloğun sonuna bir sonraki bloğun adresini tutan bir işaretçi yerleştirilir. Daha sonra bu bellek yerine ihtiyaç kalmadığı zaman, örneğin stoktaki o mala ait bilgiler silindiğinde, kullanılan hafıza alanları iade edilmektedir. Ayrıca bellek iki boyutlu değil doğrusaldır. Sıra sıra hücrelere bilgi saklanır. Belleği etkin şekilde kullanmak için veri yapılarından yararlanmak gerekmektedir. Bu sayede daha hızlı ve belleği daha iyi kullanabilen programlar ortaya çıkmaktadır.

Programlama kısmına geçmeden önce bazı kavramları açıklamakta fayda vardır. Programların çoğu birer function (*fonksiyon*) olarak yazılmıştır. Bu fonksiyonları yazarken dikkat edilmesi gereken nokta ise, bu fonksiyonların nasıl kullanılacağıdır. Fonksiyonlar genellikle bir değer atanarak kullanılırlar (parametre). Örneğin verilen nokta sayısına göre bir çokgen çizen bir fonksiyonu ele alalım. Kodu temel olarak şu şekilde olmaktadır;

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

Yukarıdaki program parçasında değer olarak `kenar` değeri atanacaktır. Mesela beşgen çizdirmek istediğimizde `cokgen_ciz(5)` olarak kullanmamız gerekir. Diğer bir örnek ise verilen `string` bir ifadenin içerisindeki boşlukları altçizgi (_) ile değiştiren bir fonksiyonumuz olsun. Örneğin `string` ifademiz "`Muhendislik Fakultesi`" ise fonksiyon sonucunda ifademiz "`Muhendislik_Fakultesi`" olacaktır. Öncelikle fonksiyonun değer olarak `string` türünde tanımlanmış "`okul`" değişkenini aldığını ve sonucu da `string` olarak tanımlanmış "`sonuc`" değişkenine attığını farz edelim. Fonksiyon tanımlaması şu şekilde olacaktır;

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

Eğer fonksiyon, boşlukları "_" ile değiştirdikten sonra yeni oluşan ifadeyi tekrar `okul` değişkenine atasaydı fonksiyon tanımlaması şu şekilde olacaktı;

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

Fark açıkça görülmektedir. Birinci programda fonksiyonun dönüş tipi yok iken, ikinci programda ise `char*` olarak dönüş tipi tanımlanmıştır. Bunun anlamı ise birinci programın `okul` değişkeninde herhangi bir değişiklik yapmayacağıdır.

Ancak ikinci programda `return` kodu ile `okul` geri döndürüldüğü için fonksiyonunun çalıştırılmasından sonra değişkenin içeriği değişecek 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 çağrısında argümanlar değere göre çağırma ile geçirilirse, argümanın değerinin bir kopyası oluşturulur ve çağırılan fonksiyona geçirilir. Oluşturulan kopyadaki değişiklikler, çağırıcıdaki orijinal değişkenin değerini etkilemez. Bir argüman referansa göre çağrıldığında ise çağırıcı, çağrılan fonksiyonun değişkenin orijinal değerini ayarlamasına izin verir. İki örnekle konuyu hatırlatalım.

Örnek 9 Swap işlemi için call by value ve call by reference yöntemiyle fonksiyonlara çağrı 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 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;
}
```