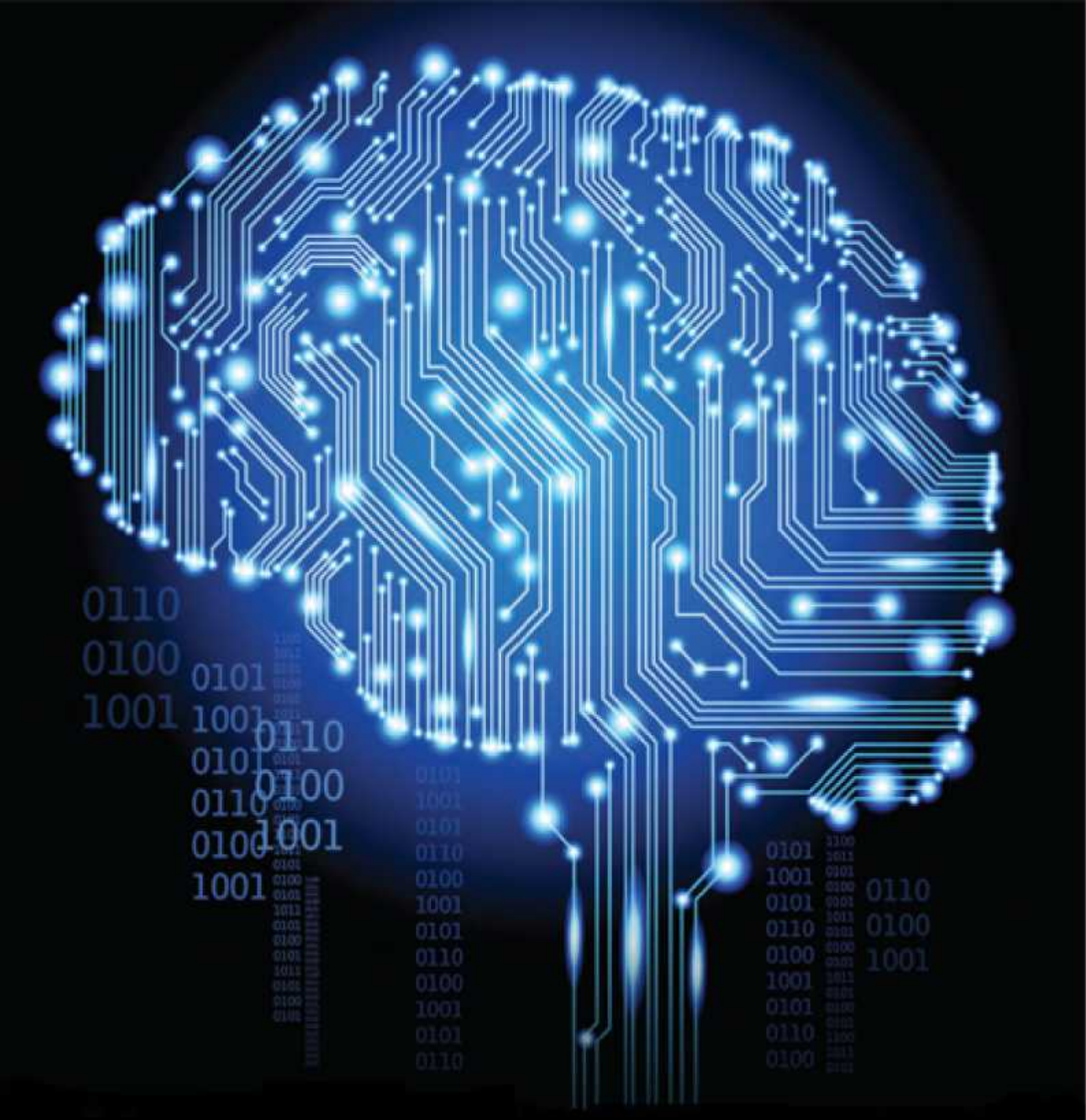


VERİ YAPILARI

DATA STRUCTURES

Asst.Prof.Dr. HAKAN KUTUCU

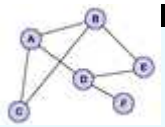
HAKAN KUTUCU

VERİ YAPILARI
(DATA STRUCTURES)

Düzenleyen
SAİM MEHMET ÖZTÜRK

KARABÜK ÜNİVERSİTESİ
Mühendislik Fakültesi Merkez Kampüsü – Karabük 2014

İçindekiler



BÖLÜM

1 Veri Tipleri

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 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.

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	3.4×10^{-38} - 3.4×10^{38}
`double`	64	1.7×10^{-308} – 1.7×10^{308}

*İşlemciye göre 16 veya 32 bitlik olabilmektedir.

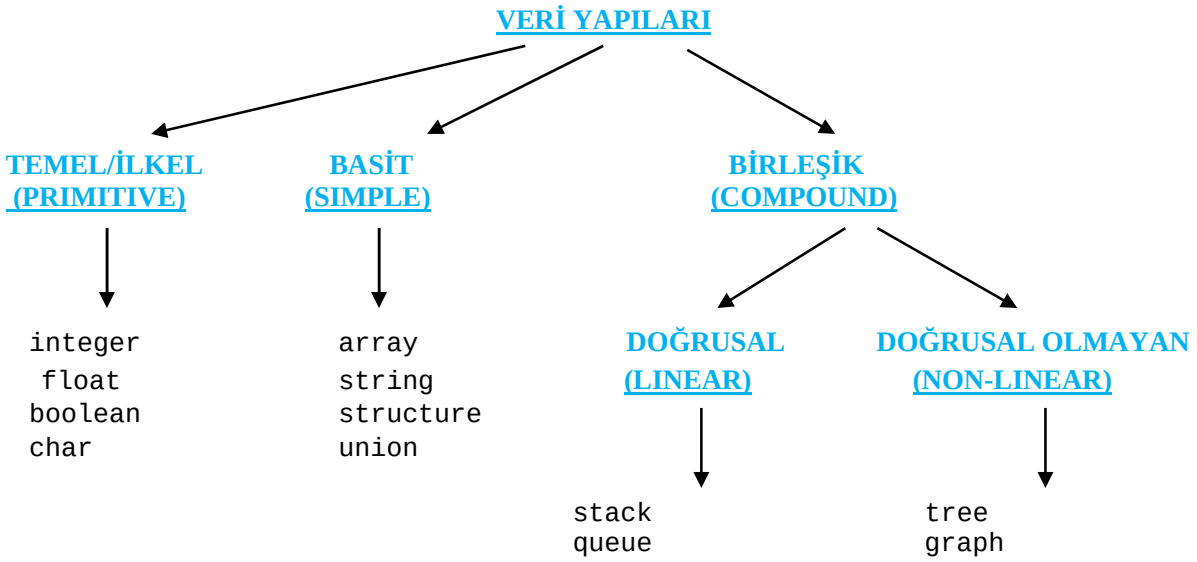
Tablo1.1 C'de veri tipleri ve tanım aralıkları.

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.

Veri Yapısı

Verileri tanımlayan veri tiplerinin, birbirleriyle ve hafızayla ilgili tüm teknik ve algoritmik özellikleridir. C dilinin Veri Yapıları Şekil l.1'deki gibi sınıflandırılabilir.



Şekil 1.1 C Dilinin veri yapıları.

Şekilden de görüleceği ve ileriki bölümlerde anlatılacağı 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.

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;

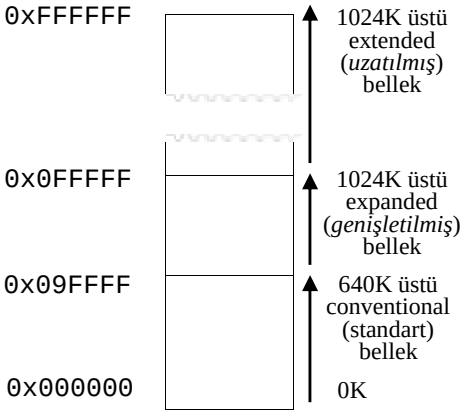
```
0100 0010 0100 0001 0100 0010 0100 0001
      16961               16961
```

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
```

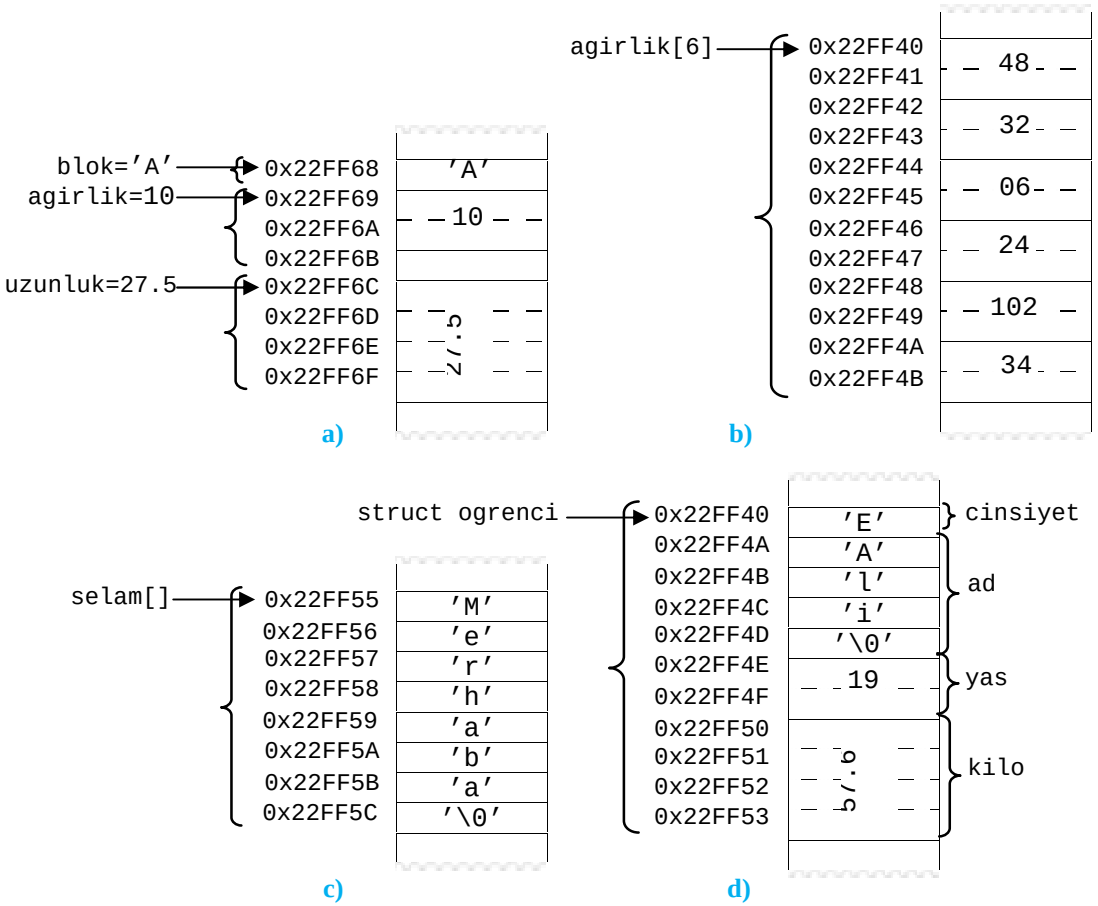
Belleğin Yapısı ve Veri Yapıları

Yapısal olarak bellek, büyüklüğüne bağlı olarak binlerce, milyonlarca `l`'er Byte (8 Bit)'lik veriler saklayabilecek biçimde tasarlanmış bir elektronik devredir. Bellek, Şekil 1.2'deki gibi her byte'ı bir hücre ile gösterilebilecek büyükçe bir tablo olarak çizilebilir.



Şekil 1.2 Belleğin adreslenmesi ve 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. Şekil 1.3'te DOS işletim sistemi için bellek haritası görülmektedir.



Şekil 1.3 Veri yapılarının bellek üzerindeki 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 Şekil 1.3 a)'daki gibi bir yerleşim gerçekleşir. İşletim sistemi değişkenleri bellekteki boş alanlara veri tiplerinin özelliklerine uygun alanlarda 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 1.3 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 1.3 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 1.3 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.

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;
}
```

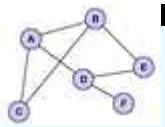
Basit veri yapılarının adreslerinin kullanımları da temel veri yapılarının kullanımlarına benzemektedir. Aşağıdaki kodlarla benzerlikler ve farklılıklar incelenmektedir.

```
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.

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.



BÖLÜM

Veri Yapıları 2

2.1GİRİŞ

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 maharet 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 2.1 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;
}
```

2.2 ÖZYİNELEMELİ FONKSİYONLAR

Özyinelemeli (*rekürsif*) fonksiyonlar kendi kendini çağıran 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 tekniğidir. Fakat her problem rekürsif bir çözüme uygun değildir. Problemin doğası ona el vermeyebilir. Rekürsif bir çözüm elde etmek için gerekli olan iki adet strateji şu şekildedir:

1. Kolayca çözülebilen bir temel durum (*base case*) tanımlamak, buna çıkış (*exitcase*) durumu da denir.
2. Yukarıdaki tanımı 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 çağıracaktır. Temel durumda ise çözüm aşikârdır.

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 şartı olmalıdır. Eğer base-case durumu sağlanıyorsa sonuç rekürsif bir uygulama olmaksızın iteratif (*özyinelemesiz*) olarak hesaplanır. Şayet base-case şartı 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 olduğu 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!  // Görüldüğü 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! = { 1          eğer n = 0
     { n.(n-1)!   eğer n > 0
```

şeklinde olacaktır.

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

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

Fonksiyona `4` rakamını gönderdiğimizi düşünelim. `4` sıfıra eşit olmadığından fonksiyon else kısmına gidecek ve `4*fact(3)` şekliyle kendini `3` rakamıyla tekrar çağıracaktır. Bu adımlar aşağıda gösterilmiştir;

```
4 * fact(3)
        ↓
 3 * fact(2)
        ↓
  2 * fact(1)
        ↓
   1 * fact(0)
        ↓
      (1)
```

Şekilde görüldüğü gibi en alttaki fonksiyon çağrısında iteratif durum gerçekleştiği için fonksiyon sonlanacak ve ilk çağrıldığı noktaya geri dönecektir. Bu esnada bellekte `fact(0)` çağrısı yerine `1` yazılacağı için o satırdaki durum `1*1` şeklini alacak ve `1` sonucu üretilecektir. Bu sefer `fact(1)` çağrısının yapıldığı yere dönecek ve yine `2*1` şeklini alıp 2 üretecektir. Sonra `fact(3)` çağrısının olduğu satırda `3*2` şeklini alıp `6` ve `fact(4)` çağrısının yapıldığı satıra döndükten sonra da `4*6` hesaplanarak `24` sayısı üretilecektir.

```
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
```

İşi biten fonksiyon bellekten silinir.

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

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

```
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 değerine 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 oluşan bir sayı dizisidir. Aşağıda klavyeden girilecek n değerine kadar olan fibonacci dizisini rekürsif olarak hesaplayan fonksiyon görülüyor. 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. Örneğin programlama dillerinde reel sayılar için `double`, tamsayılar için `int` yapısı tanımlıyken, karmaşık sayılar için böyle bir ifade yoktur. Bu yapıyı `struct` ile oluşturmak mümkündür.

Örnek 2.6 Bir `z` karmaşık sayısını ele alalım.

$z = x + iy$

real imaginary

Yapı tanımlamanın birkaç yolu vardır. Şimdi bu sayıyı tanımlayacak bir yapı oluşturalım ve bu yapıdan bir nesne tanımlayalım;

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

dediğimiz zaman `a` ve `b` birer `complex` sayı olmuşlardır. Tanımlanan `a` ve `b`'nin elemanlarına ulaşmak için nokta operatörü kullanırız. Eğer `a` veya `b`'nin birisi ya da ikisi de pointer olarak tanımlansaydı ok `(->)` operatörüyle elemanlarına ulaşacaktık. Bir örnek yapalım.

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

Şimdi de hem pointer olan hem de bir nesne olan tanımlama yapalım ve elemanlarına erişelim.

```
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 tanımlanmıştır. `p` ponter'ına ise `obj` nesnesinin adresi atanmıştır. `obj` nesnesinin elemanlarına hem `obj`'nin kendisinden, hem de `p` pointer'ından erişilebilir. `p` pointer'ından erişilirken ok `(->)` operatörü, `obj` nesnesinden erişilirken ise nokta `(.)` operatörü kullanıldığına dikkat ediniz. `p` pointer'ından erişmekle `obj` nesnesinde erişmek arasında hiçbir fark yoktur.

Örnek 2.7 İki karmaşık 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;
```

Görüldüğü gibi bir `typedef` anahtar sözcüğüyle tanımlanan `struct` yapısından hemen sonra yapı ismi tanımlanıyor. Artık bu tanımlamadan sonra nesneleri oluştururken başa `struct` yazmak gerekmeyecektir.

```
complex a, b;
```

tanımlamasıyla `complex` türden `a` ve `b` isimli nesne meydana getirilmiş olur.

2.4 VERİ YAPILARI

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

1- Matematiksel ve mantıksal modeller (*Soyut Veri Tipleri – Abstract Data Types - ADT*): Bir veri yapısına bakarken tepeden (*yukarıdan*) soyut olarak ele alacağız. Yani hangi işlemlere ve özelliklere sahip olduğuna bakacağız. Örneğin bir TV alıcısına soyut biçimde bakarsak elektriksel bir alet olduğu için onun açma ve kapama tuşu, sinyal almak için bir anteni olduğunu göreceğiz. Aynı zamanda görüntü ve ses vardır. Bu TV'ye matematiksel ve mantıksal modelleme açısından bakarsak içindeki devrelerin ne olduğu veya nasıl tasarlandıkları konusuyla ve hangi firma üretmiş gibi bilgilerle ilgilenmeyeceğiz. Soyut bakışın içerisinde uygulama (*implementasyon*) yoktur.

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

Liste;

- Herhangi bir tipte belirli sayıda elemanları saklasın,
- Elemanları listedeki koordinatlarından okusun,
- Elemanları belirli koordinattaki diğer elemanlarla değiştirsin (*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 kullanışlı 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 tanımlanmışa, eleman sayısı çarpı `int` türünün kapladığı alan kadar bellekte yer kaplayacaktır,
- Ekleme işleminde dizinin diğer elemanlarını kaydırmak gerekir. Bu işlemi yaparken dizinin boyutunun da aşılmaması gerekir,
- Silme işlemlerinde değeri boş elemanlar oluşacaktır.

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 görünüşüne bakacağız,
2- İçinde barındırdığı işlemelere 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 öğeleri arasında kurulur. Doğrusal Liste (*Linear List*) yapısı yalnızca öncelik-sonralık ilişkisini yansıtabilecek yapıdadır. Liste yapısı daha karmaşık gösterimlere imkan sağlar. Listeler temel olarak tek bağlı ve çift bağlı olmak üzere ikiye ayrılabilir. Ayrıca listelerin dairesel veya doğrusal olmasına göre de bir gruplandırma yapılabilir. Tek bağlı listelerde node'lar sadece bir sonraki node ile bağlanarak bir liste oluştururlar. Çift bağlı (*iki bağlı*) listelerde ise bir node'da hem sonraki noda hem de önceki noda bir bağlantı vardır. Bu bağlantılar Forward Link (*ileri bağ*) ve Backward Link (*geri bağ*) olarak adlandırılırlar. Doğrusal listelerde listede bulunan en son node'un başka hiçbir noda bağlantısı yoktur. Bağ değeri olarak `NULL` alırlar. Dairesel listelerde ise en sondaki node, listenin başındaki node'a bağlanmıştır. Aşağıda buna göre yapılan sınıflandırma görülmektedir.

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

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

Bağlı Listeler İle İşlemler

Bağlı listeler üzerinde;

1- Liste oluşturma,
2- Listeye eleman eklemek,
3- Listeden 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.

2.6 TEK BAĞLI DOĞRUSAL LİSTELER

Tek bağlı doğrusal liste, öğelerinin arasındaki ilişki (*Logical Connection*)'ye göre bir sonraki öğenin bellekte yerleştiği yerin (*Memory Location*) bir gösterge ile gösterildiği yapıdır. Bilgisayar belleği doğrusaldır. Bilgiler sıra sıra hücrelere saklanır. Her bir bilgiye daha kolay ulaşmak 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.

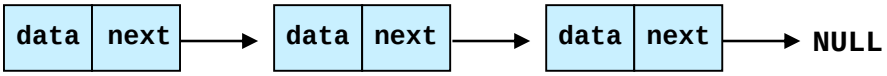
Tek bağlı listelerin genel yapısı aşağıda verilmiştir. Konu anlatılırken daima bu temel yapı kullanılacağından unutmamalısınız.

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

Bağlı listeler içerisindeki düğümlerin yukarıdaki tanımlamayla iki öğesinin olduğu görülüyor. Birinci öğe olan `data`, her türden veri içerebilir, örneğin telefon numaraları, TC kimlik numaraları vb. gibi. Biz `int` türden bir nesneyi yeterli bulduk. İkinci öğe olan `next`, bir bağlı listede mutlaka bulunması gereken bir öğedir. Dikkat edilirse `struct node` tipinde bir işaretçidir.

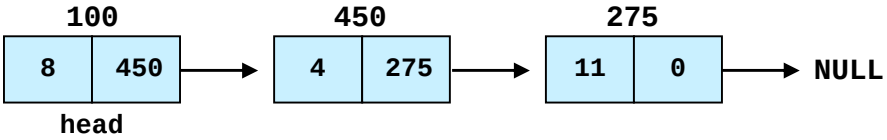
Şekil 2.1 Tek Bağlı Listelerde bir node'un mantıksal yapısı.

Tek bağlı listelerin yapısında bulunan `node` tipindeki `*next` işaretçisi, rekürsif fonksiyonlarla karıştırılmamalıdır. Burada `next`, bir sonraki `node` türden düğümün adresini gösterecektir.



Şekil 2.2 Tek bağlı listeler.

Alttaki şekilde her çiftli kutucuk liste yapısını temsil etmektedir. Bu kutucukların üstündeki numaralar ise bellekte bulundukları yerin adresidir. Burada önemli olan, `next` göstericisinin değerlerinin, yani düğüm adreslerinin sıralı olmayışıdır. İlk düğümün adresi 100, ikincisinin 450 ve üçüncüsünün ise 275'tir. Yani bellekte neresi boşsa o adresi almışlardır. Oysa dizilerde tüm elemanlar sıralı bir şekilde bellekte yer kaplıyordu.



Şekil 2.3 Liste yapısı ve bellekteki adreslerinin mantıksal gösterimi.

Listenin ilk elemanı genellikle `head` olarak adlandırılır. `head`'den sonra diğer elemanlara erişmek kolaydır. Bazı kaynaklarda listenin sonundaki elemanın ismi `tail` olarak adlandırılmıştır. Fakat biz bu ismi notlarımızda kullanmayacağız.

Tek Bağlı Doğrusal Liste Oluşturmak ve Eleman Eklemek

Bu örnekte ilk olarak listeyi oluşturacağız, ardından eleman ekleme yapacağız.

```
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? Tabi ki hayır! Elbette ki bunu yapmanın daha kolay bir yolu var.

Tek Bağlı Doğrusal Listenin Başına Eleman Eklemek

Bir fonksiyonla örnek verelim. Siz isterseniz `typedef` anahtar sözcüğünü kullanarak sürekli `struct` yazmaktan kurtulabilirsiniz fakat biz akılda kalıcı olması açısından `struct`'ı kullanmaya devam edeceğiz.

```
// Tek bağlı doğrusal 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 değil
     aslında. Çünkü boş ise zaten head=NULL olacaktır ve temp olan ilk düğümün
     next'i NULL gösterecektir. */
    head = temp; /* head artık temp'in adresini tutuyor yani eklenen düğüm
                    listenin başı oldu. */
    return head;
}
```

Tek Bağlı Doğrusal Listenin Sonuna Eleman Eklemek

Listenin sonuna ekleme yapabilmek için liste sonunu bilmemiz gerekiyor. Listede eleman olduğunu varsayıyoruz. Liste sonunu bulabilmek içinse bu liste elemanları üzerinde tek tek ilerlemek gerektiğinden `head`'in adresini kaybetmemek için bir değişken oluşturacağız ve `head`'in adresini bu değişkene atayacağız.

```
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 kullanılabileceğini unutmayınız. */
    temp -> data = key;
    temp -> next = NULL;
    struct node *temp2 = head;
    /* Aşağıdaki while yapısı traversal(dolaşma) olarak adlandırılır */
    while(temp2 -> next != NULL)
        temp2 = temp2 -> next;
    temp2 -> next = temp;
    return head;
}
```

Tek Bağlı Doğrusal Liste Elemanlarının Tüm Bilgilerini Yazdırmak

Listedeki elemanların adreslerini, data kısımlarını ve sonraki düğüm adresini ekrana basan `listinfo` ismindeki 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 Bağlı Doğrusal Listenin Elemanlarını Yazdırmak

Fonksiyon sadece ekrana yazdırma işi yapacağından `void` olarak tanımlanmalıdır. Liste boşsa ekrana listede eleman olmadığına 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 düğüm yazılmazdı
        printf("%d\n", temp2 -> data);
        temp2 = temp2 -> next;
    }
}
```

Şüphesiz elemanları yazdırmak için özyinelemeli bir fonksiyon da kullanılabilirdi.

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

SORU: Yukarıdaki fonksiyon aşağıdaki 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 Bağlı Doğrusal Listenin Elemanlarını Saymak

Listenin elemanlarını saymak için `int` tipinden bir fonksiyon oluşturacağız. Ayrıca listede eleman olup olmadığını da kontrol etmeye gerek yoktur. Çünkü eleman yok ise while döngüsü hiç çalışmayacak ve 0 değerini döndürecektir.

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

Bu işlemi özyinelemeli yapmak istersek:

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

Tek Bağlı Doğrusal Listelerde Arama Yapmak

Bu fonksiyon ile liste içinde arama yapılmaktadır. Eğer aranan bilgi varsa, bulunduğu `node`'un adresiyle geri döner. Bu fonksiyon bulundu ise true bulunmadı ise false döndürecek şekilde de düzenlenebilir.

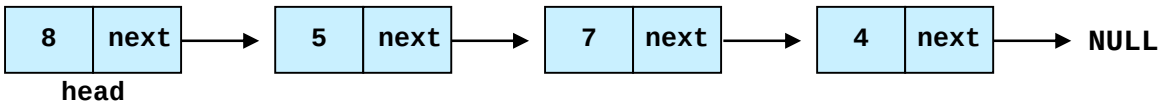
```
struct node* locate(struct node* head, int key) {
    struct node* locate = NULL;
    while(head != NULL)
        if(head -> data != key)
            head = head -> next;
        else {
            locate = head;
            break;
        }
    return(locate);
}
```

Tek Bağlı Doğrusal Listelerde İki Listeyi Birleştirmek

`list_1` ve `list_2` adındaki iki listeyi birleştirmek için `concatenate` fonksiyonunu kullanabiliriz.

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

Tek Bağlı Doğrusal Listelerde Verilen Bir Değere Sahip Düğümü Silmek



Şekil 2.4 Tek bağlı listelerin mantıksal yapısı.

Tek bağlı listelerde verilen bir düğümü silme işlemi, o düğümün başta ya da ortada olmasına göre farklılık gösterir. İlk düğüm silinmek istenirse ikinci elemanın adresini yani `head`'in `next` işaretçisinin tuttuğu adresi `head`'e atayarak baştaki eleman silinebilir. Eğer ortadan bir düğüm silinmek istenirse bir önceki düğümü, silinmek istenen düğümden bir sonraki düğüme bağlamak gerekir. Şimdi `_remove` ismini vereceğimiz 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 düğüm silinecek mi diye kontrol ediliyor.
        head = head -> next; // head artık bir sonraki eleman.
        free(temp);
    }
    else if(temp -> next == NULL) { // Listede tek düğüm bulunabilir.
         printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
         return head;
    }
    else {
        while(temp -> next -> data != key) {
            if(temp -> next -> next == NULL) {
                printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
                return head;
            }
            temp = temp -> next;
        }
        struct node *temp2 = temp -> next;
        temp -> next = temp -> next -> next;
        free(temp2);
    }
    return head;
}
```

Tek Bağlı Doğrusal Listelerde Verileri Tersten Yazdırmak

Tek bağlı listenin elemanlarını tersten yazdıran `print_reverse` adlı bir fonksiyon yazalım. Bu fonksiyonu yazarken her veriyi yeni bir listenin başına ekleyeceğiz ve böylece ilk listenin tersini elde etmiş olacağız. Bunun için `addhead` adlı fonksiyonu kullanacağız.. `print_reverse` fonksiyonunda `struct node*` türden yeni bir değişken daha tanımlayacağız ve `head`'in elemanlarını bu yeni listenin sürekli başına ekleyerek verileri ters bir biçimde sıralayacağız ve yazdırma işlemini gerçekleştireceğiz. Aslında tersten yazdırma işini rekürsif olarak yapan bir fonksiyon daha önce SORU olarak sorulmuş 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 değişkeni
```

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

Tek Bağlı Doğrusal 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 yer işgal 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 değilse
        temp2=head;
        head = head->next;
        free(temp2);
    }
    return head;
}
```

SORU: destroy fonksiyonunu özyinelemeli olarak yazınız.

Main Fonksiyonu İçinde Tanımladığımız Tüm Fonksiyonların Çağırılması

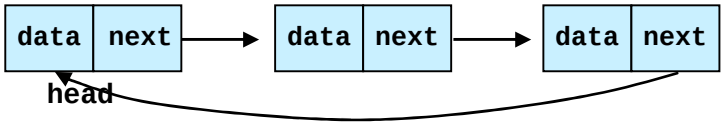
Yukarıda tanımladığımız fonksiyonların çalıştırılması için tüm fonksiyonlar, struct tanımlaması dahil aşağıdaki main() fonksiyonu üzerinde yazılır.

```
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 BAĞLI DAİRESEL (Circle Linked) LİSTELER

Tek bağlı dairesel listelerde, doğrusal listelerdeki birçok işlem aynı mantık ile benzer şekilde uygulanır; fakat burada dikkat edilmesi gereken nokta, dairesel bağlı listede son elemanının `next` işaretçisi `head`'i göstermektedir..



Şekil 2.5 Dairesel bağlı listeler.

Tek Bağlı Dairesel Listelerde Başa Eleman Eklemek

Tek bağlı listelerde yaptığımızdan farklı olarak `head`'in global olarak tanımlandığı varsayılmıştır. Liste yoksa oluşturuluyor, eğer var ise, `struct node*` türden `temp` ve `last` düğümleri oluşturularak `last`'a `head`'in adresi atanıyor (*`head`'in adresini kaybetmememiz gerekiyor*). `temp`'in `data`'sına parametre değişkeninden gelen veri aktarıldıktan sonra `last` döngü içerisinde ilerletilerek listenin son elemanı göstermesi sağlanıyor. `temp` `head`'i gösterecek şekilde atama yapıldıktan sonra listenin son elemanını gösteren `last`'ın `next` işaretçisine de `temp`'in adresi atanıyor. Şu anda `last` eklenen verinin bulunduğu düğümü gösteriyor değil mi? `temp`'in `next` işaretçisi ise `head`'i gösteriyor. `head`'e `temp` atanarak işlem tamamlanmış oluyor. **DİKKAT!** 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 Bağlı Dairesel Listelerde Sona Eleman Eklemek

Fonksiyon, başa ekleme fonksiyonuna çok benzemektedir. Listenin `NULL` olup olmadığı 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;
    }
}
```

Görüldüğü gibi başa ekleme fonksiyonunun sonundaki `head = temp;` satırını kaldırmak yeterlidir. Aynı fonksiyonu aşağıdaki şekilde de yazabilirdik. Burada `temp` isminde bir değişkene ihtiyacımızın olmadığını 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 mantığı kavramaya çalışınız. Ancak çok miktarda alıştırma yaptıktan sonra iyi bir pratik kazanabilirsiniz.

Tek Bağlı Dairesel Listelerde İki Listeyi Birleştirmek

`concatenate` fonksiyonu tek bağlı dairesel listelerde iki listeyi verilen ilk listenin sonuna ekleyerek birleştiren fonksiyon olarak tanımlanacaktır. Bu fonksiyonun yazımında önemli olan doğru işlem sırasını takip etmektir. Eğer öncelikli yapılması gereken bağlantılar sonraya bırakılırsa son `node`'un bulunmasında sorunlar çıkacaktır. `list_1` boş ise `list_2`'ye eşitleniyor. Eğer boş değilse her iki listenin de `last()` fonksiyonuyla son elemanları bulunuyor ve `next` işaretçilerine bir diğerinin gösterdiği adres değeri atanıyor.

```
// list_1 listesinin sonuna list_2 listesini eklemek
void concatenate(struct node*& list_1, struct node* list_2){ //parametrelere dikkat
    if(list_l == NULL)
        list_1 = list_2;
    else {
       // Birinci listenin son düğümünü 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 bağlandı
        // İkinci listenin son düğümünü 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 bağlandı
     }
}
```

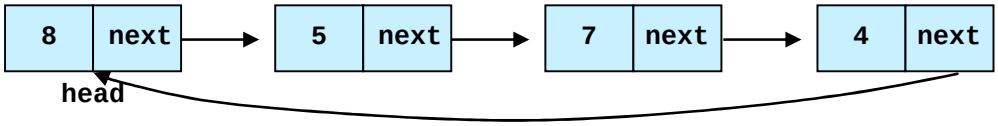
Tek Bağlı Dairesel Listede Arama Yapmak

Bu fonksiyon ile liste içinde arama yapılmaktadır. `node`'lar taranarak `data`'lara bakılır. Eğer aranan bilgi varsa, bulunduğu `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);
    return(locate);
}
```

Tek Bağlı Dairesel Listelerde Verilen Bir Değere Sahip Düğümü Silmek

Silme işlemine dair fonksiyonumuzu yazmadan önce tek bağlı dairesel listelerin mantıksal yapısını tekrar gözden geçirmekte fayda vardır. Bu liste yapısında son düğümün `next` işaretçisi `head`'i gösteriyor ve dairesel yapı sağlanıyor.



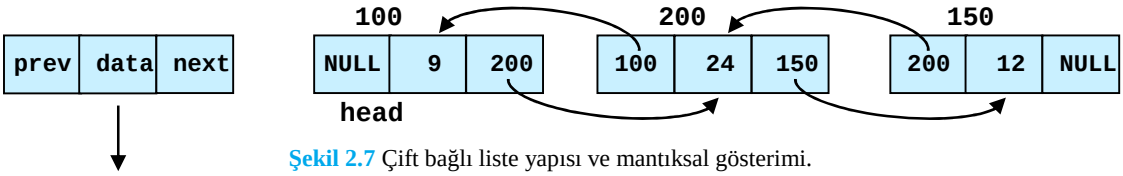
Şekil 2.6 Tek bağlı listelerin mantıksal yapısı.

Tek bağlı dairesel listelerde verilen bir değere sahip düğümü silme işlemi için `deletenode` isimli bir fonksiyon yazacağız. Fonksiyonda liste boş ise hemen fonksiyondan çıkılır ve geriye `false` değeri döndürülür. Eğer silinecek düğüm ilk düğüm ise daha önce yaptığımız gibi ilk düğüm listeden çıkarılır. Ancak burada listenin son düğümünü yeni head düğümüne bağlamak gereklidir. Bu yüzden önce son düğüm 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 düğüm 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 düğüm bulunabilir.
         printf("Silmek istediginiz veri bulunmamaktadir.\n\n");
    }
    else {
        while(temp -> next -> data != key) {
            if(temp -> next -> next == NULL) {
                printf("Silmek istediginiz veri bulunmamaktadir.\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 bağlı listelerin mantıksal yapısı Şekil 2.7'de gösterilmiştir. Her düğümün `data` adında verileri tutacağı bir değişkeni ile kendinden önceki ve sonraki düğümlerin adreslerini tutacak olan `prev` ve `next` isminde iki adet işaretçisi vardır. Listenin başını gösteren işaretçi `head` yapı değişkenidir. Şekilde `head`'in adresi `100`'dür ve `head`'in `prev` işaretçisi herhangi bir yeri göstermediğinden `NULL` değer içermektedir. `next` işaretçisi ise bir sonraki düğümün adresi olan `200` değerini içermektedir. İkinci düğümün `prev` işaretçisi `head`'in adresi olan `100` değerini tutmakta, `next` işaretçisi ise son düğümün adresi olan `150` değerini tutmaktadır. Nihayet son düğümün `prev` işaretçisi kendinden önceki düğümün adresini yani `200` değerini tutmakta ve `next` işaretçisi ise `NULL` değer içermektedir.



Şekil 2.7 Çift bağlı liste yapısı ve mantıksal gösterimi.

Çift bağlı listelerin struct yapısı aşağıda verilmiştir;

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

Çift Bağlı Doğrusal Listenin Başına Eleman Eklemek

`head` değişkeninin 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;
    }
}
```

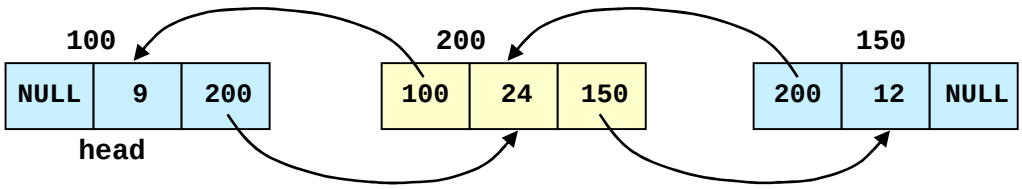
Çift Bağlı Doğrusal Listenin Sonuna Eleman Eklemek

```
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 Bağlı Doğrusal Listelerde Araya Eleman Eklemek

```
// head listesinin n. düğümünün hemen ardına other_node düğümünü 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 Bağlı Doğrusal Listelerde Verilen Bir Değere Sahip Düğümü Silmek

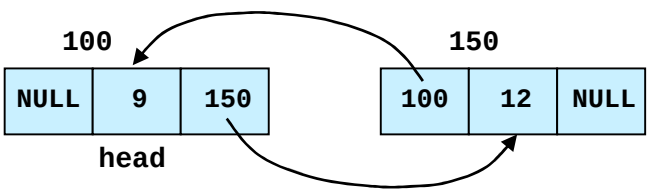


Şekil 2.8 Silinmek istenen ortadaki düğüm sarı renkte gösterilmiştir.

Silme işlemi diğer liste türlerine göre biraz farklılık göstermektedir. İlk düğüm silinmek istenirse `head`'in `next` işaretçisinin tuttuğu adresi `head`'e atadıktan sonra `prev` işaretçisinin değerini `NULL` yapmamız gerekir. Sonra `free()` fonksiyonuyla baştaki eleman silinebilir. Eğer ortadan bir düğüm silinmek istenirse, silinecek düğümün üzerinde durup bir önceki düğümü, silinmek istenen düğümden bir sonraki düğüme bağlamak gerekir. Silinecek düğüm Şekil 2.8'de görüldüğü gibi ortadaki düğüm olsun. `double_linked_remove` isimli fonksiyonumuzu yazarak konuyu kavrayalım.

```
void double_linked_remove(int key) {
    struct node *temp = head;

    if(head -> data == key) { // silinecek değerin ilk düğümde olması durumu.
        head = head -> next;
        head -> prev = NULL;
        free(temp);
    }
    else {
        while(temp -> data != key)
            temp = temp -> next;
        temp -> prev -> next = temp -> next;
        /* silinecek düğümden bir önceki düğümün next işaretçisi, şimdi silinecek
           düğümden bir sonraki düğümü gösteriyor. */

        if(temp -> next != NULL) // silinecek düğüm son düğüm değilse
            temp -> next -> prev = temp -> prev;
            /* silinecek düğümden bir sonraki düğümün prev işaretçisi, şimdi
               silinecek düğümden bir önceki düğümü gösteriyor. */
        free(temp);
    }
}
```

Listenin, ortada bulunan düğümü silindikten sonraki görünümünü Şekil 2.9'da görüyorsunuz.



Şekil 2.9 Silme işleminden sonra yeni listenin görünümü.

Çift Bağlı Doğrusal 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 Bağlı Doğrusal Listede Karşılaştırma Yapmak

Verilen `node`'un bu listede var olup olmadığını kontrol eden fonksiyondur. Fonksiyon eğer 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 doğruysa 1, değilse 0 geri döndürülür.
}
```

Verilen örneklerdeki fonksiyonlar, varsayılan bir listenin yahut global tanımlanmış `head` değişkeninin varlığı, yine global olarak tanımlanmış yapıların olduğu kabul edilerek yazılmıştır. Eğer bu kabulümüz olmasaydı şüphesiz kontrol ifadelerini de içeren uzun kod satırları meydana gelirdi.

Çift Bağlı Doğrusal Listelerin Avantajları ve Dezavantajları

Avantajları

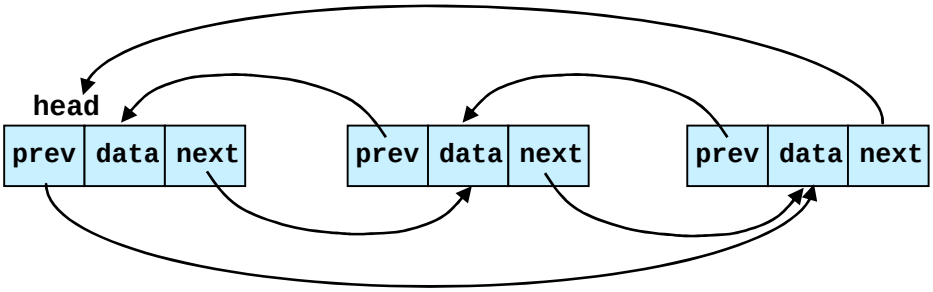
- Her iki yönde gezilebilir,
- Ekleme,Silme gibi bazı işlemler daha kolaydır.

Dezavantajları

- Bellekte daha fazla yer kaplar,
- Her düğümün `prev` ve `next` adında iki işaretçisi olduğu için liste işlemleri daha yavaştır,
- Hata olasılığı yüksektir. Örneğin düğümlerin `prev` işaretçisinin bir önceki düğüme bağlanması ya da `next` işaretçisinin bir sonraki düğüme bağlanması unutulabilir.

2.9 ÇİFT BAĞLI DAİRESEL(Double Linked) LİSTELER

İki bağlı doğrusal listelerde listenin birinci düğümünün `prev` değeri ve sonuncu düğümünün `next` değeri `NULL` olmaktaydı. Ancak iki bağlı dairesel listelerde listenin birinci düğümünün `prev` değeri sonuncu düğümünün `data` kısmını, sonuncu düğümünün `next` değeri de listenin başını (*yani listenin ismini*) göstermektedir. Fonksiyonlarda bununla beraber değişmektedir.



Şekil 2.10 Çift bağlı dairesel listeler.

Çift Bağlı Dairesel Listelerde Başa Düğüm Eklemek

Listenin başına bir düğüm 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 bağlı ve dairesel olduğu için son eleman head->prev dir.
        head->prev->next=temp;
        temp->next=head;
        temp->prev=head->prev;
        head->prev=temp;
        head = temp;
    }
}
```

Çift Bağlı Dairesel Listenin Sonuna Eleman Eklemek

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

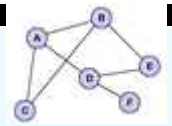
```
void addlast(struct node* temp, struct node*&head) {
    if(!head)
        head = temp;
    else {
        temp -> next = last(head) -> next;
        temp -> prev = last(head);
        last(head) -> next = temp; // last fonksiyonu ile son düğüm bulunuyor
        head -> prev = temp;
    }
}
```

Çift Bağlı Dairesel Listelerde İki Listeyi Birleştirmek

```
// list_1 listesinin sonuna list_2 listesini eklemek
void concatenate(struct node*& list_1, struct node* list_2){ //parametrelere dikkat
    if(list_l == NULL)
        list_1 = list_2;
    else {
       // Birinci listenin son düğümünü 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 bağlandı
        list2->prev=last; //İkinci listenin başı birinci listenin sonuna bağlandı
        // İkinci listenin son düğümünü 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 bağlandı
        list1->prev=last; //Birinci listenin başı ikinci listenin sonuna bağlandı
    }
}
```

Çift Bağlı Dairesel Listelerde Araya Eleman Eklemek

```
// head listesinin n. düğümünün hemen ardına other_node düğümünü 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;
}
```

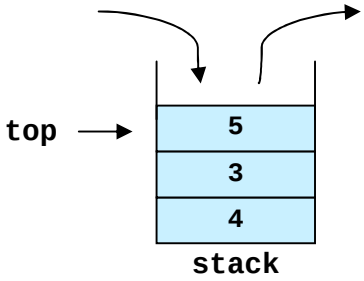


BÖLÜM

Yığınlar (*Stacks*) 3

3.1 YIĞINLARA (*Stacks*) GİRİŞ

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



Şekil 3.1 Bir stack'in mantıksal yapısı.

Şekilde görüldüğü gibi, nesneler *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;

- şarjör,
- yemekhanedeki tepsiler,
- 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 şeklinde, dinamik olarak tanımlanırken ise bağlı liste biçiminde tanımlanabilir. Şimdi stack'lerin C programlama dilini kullanarak bilgisayarlardaki implementasyonunun nasıl gerçekleştirildiğini görelim.

3.2 STACK'LERİN DİZİ (*Array*) İMPLEMENTASYONU

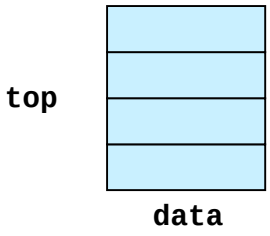
Stack'lerin bir boyutu (*kapasitesi*) olacağından önce bu boyutu tanımlamalıyız.

```
#define STACK_SIZE 4
```

C derleyicilerinin `#define` önişlemci komutunu biliyor olmalısınız. C programlama dilinde # ile başlayan bütün satırlar, önişlemci 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 değil `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 değiştirme zahmetinden kurtaracaktır. Şimdi bir `stack`'in genel yapısını tanımlayalım;

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

Bu tanımlama ile meydana gelen durum aşağıdaki şekilde görülmektedir.



Şekil 3.2 LIFO ihtiyacı için kullanılacak veri yapısı.

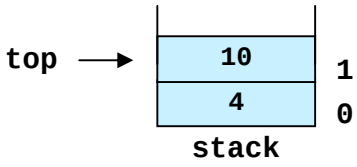
Dikkat ederseniz `top` değişkeni şu an için herhangi bir `data` 'nın indisini tutmuyor. C derleyicisi tarafından bir çöp değer atanmış durumdadır. `stack`'e ekleme işlemi yapan `push` isimli fonksiyonumuzu yazalım.

Stack'lere Eleman Ekleme İşlemi (push)

Bir `stack`'e eleman ekleme fonksiyonu `push` olarak bilinir ve bu fonksiyon, `stack`'i ve eklenecek elemanın değerini parametre olarak alır. Geri dönüş değeri olmayacağı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için yığının 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 değişkeni en son eklenen elemanın indis değerini tutmaktadır.

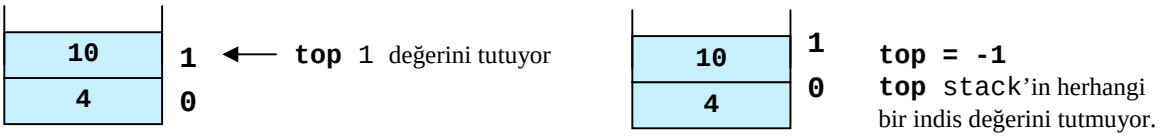


Şekil 3.3 `top` değişkeni `1` indis değerini tutuyor.

Bir Stack'in Tüm Elemanlarını Silme İşlemi (`reset`)

`stack`'i resetlemek için `top` değişkeninin değerini `-1` yapmamız yeterlidir. Bu atamadan sonra `top` herhangi bir indis değeri tutmuyor demektir. Aslında veriler silinmedi değil mi? Bu veriler hala hafızada bir yerlerde tutuluyor. Lakin `stack`'e hiç ekleme yapılmamış bile olsa `stack`'lerin dizi ile implementasyonu bellekte yine de yer işgal edecekti. Yeni eklenecek veriler ise öncekilerinin üzerine yazılacaktır. Şimdi `reset` ismindeki fonksiyonumuzu yazalım.

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



Şekil 3.4 `Stack`'lerde resetleme işlemi.

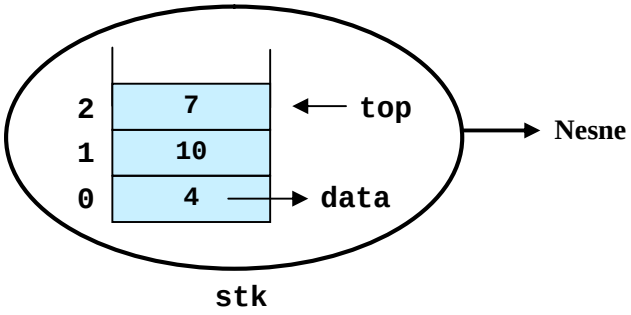
Stack'lerden Eleman Çıkarma İşlemi (pop)

`pop` fonksiyonu olarak bilinir. Son giren elemanı çıkarmak demektir. Çıkarılan elemanın indis değeriyle geri döneceği 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 değeriyle geri dönüyor
    }
}
```

`else` kısmı şu tek satır kodla da yazılabilirdi. Fakat bu kez geri dönüş değerinin `stack`'den çıkarılan verinin kendisi olduğuna dikkat ediniz.

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



Şekil 3.5 `stk` nesne modeli.

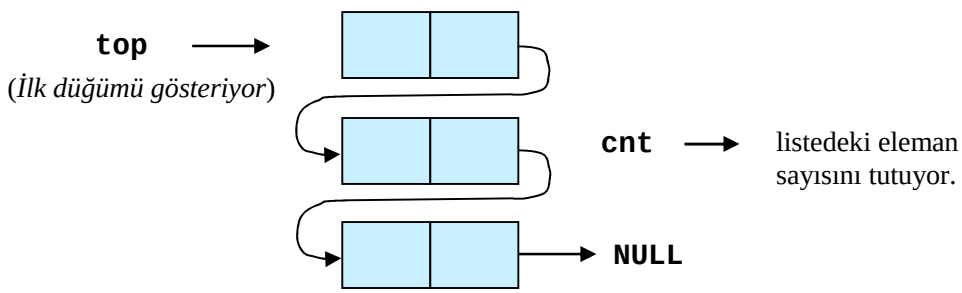
Aşağıda `top()` ve `pop()` fonksiyonlarının `main()` içerisinde ki kullanımını görüyorsunuz. `stack` yapısının ve fonksiyon tanımlamalarının `main()` bloğu içinde yapıldığını varsayıyoruz.

```
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 yapı işaretçileriyle beraber bulunur. Mantıksal yapısını daha önce gördüğümüz bağlı listelerin mantıksal yapısının üst üste sıralanmış şekli gibi düşünebilirsiniz.



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

Dizi uygulamasının verdiği maksimum genişlik sınırı kontrolü bu uygulamada yoktur. Tüm fonksiyonlar sabit zamanda gerçekleşir. Çünkü fonksiyonlarda `stack`'in genişliği 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 bağlı doğrusal liste kullanarak veri yapısını yazalım.

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

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

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

C'de enumaration (*numaralandırma*) konusunu hatırlıyor olmalısınız. Biz de ilk olarak enumaration 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 direk `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 değil ise `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` değişkeninin değeri, `stack`'in boyutunu sınırlandıran `STACK_SIZE` değişkenine eşitse dolu demektir ve fonksiyondan geriye `true` döndürülür. Eşit değilse geri dönüş değeri `false` olacaktır.

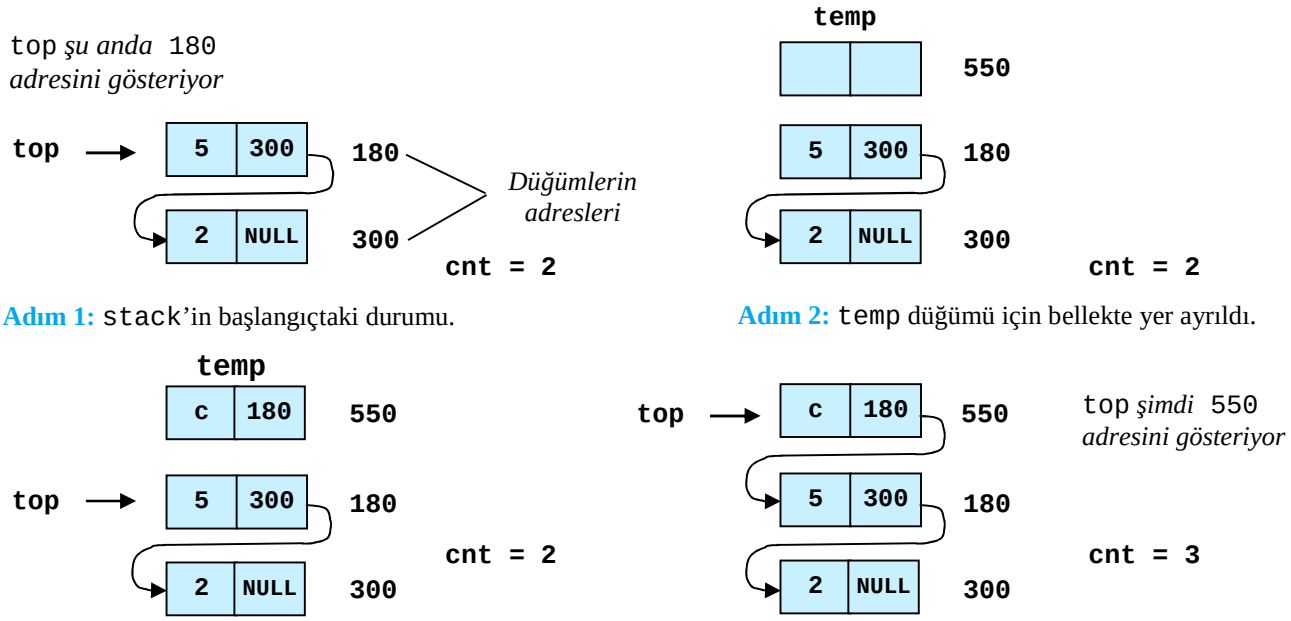
```
boolean isFull(stack *stk) {
    return(stk -> cnt == STACK_SIZE);
}
```

Stack'lere Yeni Bir Düğüm Eklemek (push)

Tek bağlı liste uygulamasında da `stack`'lerde ekleme işlemini 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 bağlı 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 düğüm ekleneceğinden dolayı fonksiyon içerisinde `struct node` türden bir yapı oluşturmak 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 bağlı listelerde eleman ekleme işleminden tek farkı `stk -> cnt++` ifadesidir. Yapılan işlemler aşağıdaki şekilde adım adım belirtilmiştir.



Adım 1: `stack`'in başlangıçtaki durumu.

Adım 2: `temp` düğümü için bellekte yer ayrıldı.

Adım 3: Yeni düğüme veriler atandı.

Adım 4: `top` artık listenin başını gösteriyor.

Şekil 3.7 Bağlı liste kullanılarak oluşturulmuş bir `stack`'e yeni bir düğüm eklemek.

Stack'lerden Bir Düğümü Silmek (pop)

Yine dizi uygulamasında olduğu gibi `pop` fonksiyonu olarak bilinir. Silinecek düğümü belleğe geri kazandırmak için `struct node` türden `temp` adında bir işaretçi tanımlanarak silinecek düğümün adresi bu işaretçiye atanır. `top` göstericisine ise bir sonraki düğümün adresi atanır ve `temp` düğümü `free()` fonksiyonuyla silinerek belleğe geri kazandırılır. Eleman sayısını tutan `cnt` değişkeninin değeri de `1` azaltılır. Fonksiyon silinen düğümün verisiyle geri döneceği için türü `int` olmalıdır.

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

Fonksiyon, eğer `stack` boş değilse düğüm silme işlemini gerçekleştiriyor ve silinen düğümün 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 değilse en üstteki düğümün verisini geri döndüren bir fonksiyondur. Geri dönüş değeri 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 Değerlerini Vermek (`initialize`)

Önemli bir fonksiyondur. Değişkenlere bir değer ataması yapılmadığı zaman çoğu C derleyicisi bu değişkenlere, çöp değer diye de tabir edilen rastgele değerler atayacaktır. Yığın işlemlerinden önce mutlaka bu fonksiyonla başlangıç değerleri 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;

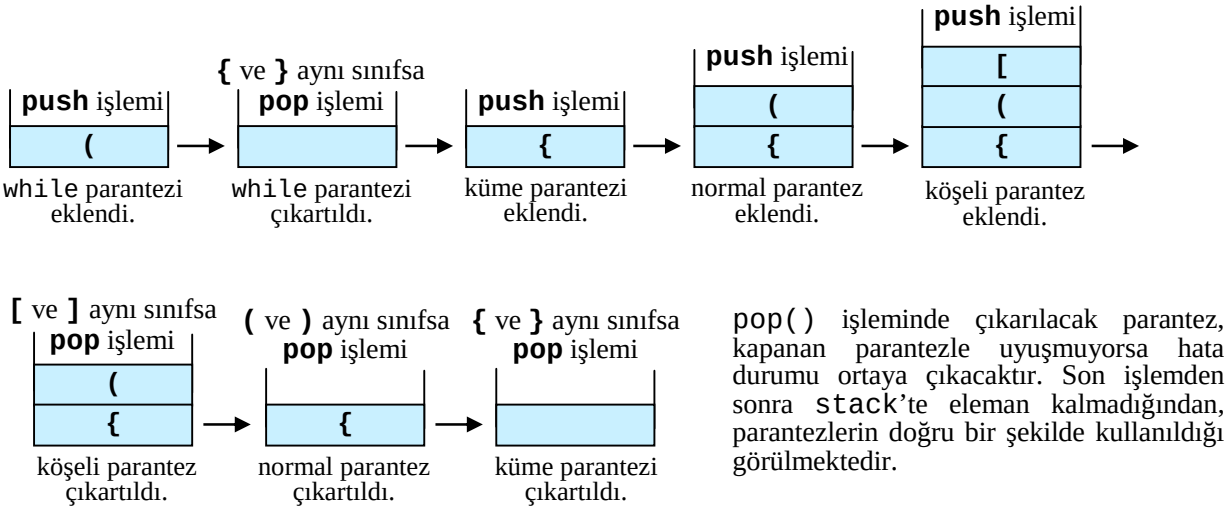
- Rekürsif olarak tanımlanan bir fonksiyon çalışırken hafıza kullanımı bu yöntem ile ele alınır,
- `(, {, [, ], }, )` ayıraçlarının `C/C++` derleyicisinin kontrollerinde,
- `postfix → infix` dönüştürmelerinde,
- Yazılım uygulamalarındaki Parsing ve Undo işlemlerinde,
- Web browser'lardaki Back butonu (*önceki sayfaya*) uygulamasında,
- Ayrıca, mikroişlemcinin içyapısında **`stack`** adı verilen özel hafıza alanı ile mikroişlemci arasında, bazı program komutları ile (*`push` ve `pop` gibi*), bir takım işlemlerde (*alt program çağırma ve kesmeler gibi*), veri transferi gerçekleşir. Mikroişlemcinin 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 çağırıldığı yere 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 yapıldığı alttaki şekilde adımlar halinde görülüyor. Bu uygulamada yukarıdaki kod parçasında bulunan bir parantez açıldığında `push()` fonksiyonuyla `stack`'e ekleme, kapatıldığında ise `pop()` fonksiyonuyla çıkarma yapacağız.



`pop()` işleminde çıkarılacak parantez, kapanan parantezle uyuşmuyorsa hata durumu ortaya çıkacaktır. Son işlemden sonra `stack`'te eleman kalmadığından, parantezlerin doğru bir şekilde kullanıldığı görülmektedir.

Şekil 3.8 Parantez kontrolünün şekilsel gösterimi.

3.4 INFIX, PREFIX VE POSTFIX NOTASYONLARI

Bilgisayarlarda infix yazım türünün çözümlenmesi zordur. Acaba `x=a/b-c+d*e-a*c` şeklindeki bir ifadeyi çözümlerken, `((4/2)-2)+(3*3)-(4*2)` gibi bir ifadenin değerini hesaplarken ya da `a/(b-c)+d*(e-a)*c` gibi parantezli bir ifadeyi işlerken derleyiciler sorunun üstesinden nasıl geliyor? `32*(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 ifadede farklı önceliklere sahip operatörler yazılma sırasıyla işlenirse ifade yanlış sonuçlandırılabilir. Örneğin `3+4*2` ifadesi `7*2=14` ile sonuçlandırılabileceği gibi `3+8=11` ile de sonuçlandırılabilir. Bilgisayarlarda infix yazım türünün çözümlenmesi 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: Alışa geldiğimiz ifadeler infix şeklindedir. Operatörlerin işlenecek operandlar arasına yerleştirildiği gösterim biçimidir. Bu gösterimde operatör önceliklerinin değiştirilebilmesi için parantez kullanılması şarttır. Örneğin 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. Örneğin `2+4*6` ifadesi infix notasyonundadır ve prefix notasyonunda `+2*46` şeklinde gösterilir. Benzer biçimde `(2+4)*6` ifadesi `*+246` şeklinde gösterilir. Görüldüğü gibi prefix notasyonunda işlem önceliklerinin sağlanması için parantezlere ihtiyaç duyulmamaktadır.

Postfix notasyonu: Postfix notasyonunda (RPN, *reverse polish notation*) ise önce operandlar ve ardından operatör yerleştirilir. Aynı örnek üzerinden devam edersek; infix notasyonundaki `2+4*6` ifadesi prefix notasyonunda `2 4 6 * +` şeklinde, benzer biçimde `(2+4)*6` ifadesi de `2 4 + 6 *` şeklinde gösterilir. Yine prefix'te olduğu 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*) yığını ile birleştirilmiş 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 alıştırma yapıldığında, iç içe birçok parantez içeren uzun ifadeleri, terimlerin nasıl gruplandığını bile düşünmeden, hızlı bir şekilde hesaplamak mümkündür.

İşlem önceliği;

1- Parantez içi
2- Üs alma
3- Çarpma/Bölme
4- Toplama Çıkarma

Aynı önceliğe sahip işlemlerde sıra soldan sağa (→) doğrudur. Yalnız üs almada sağdan sola doğru işlem 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.

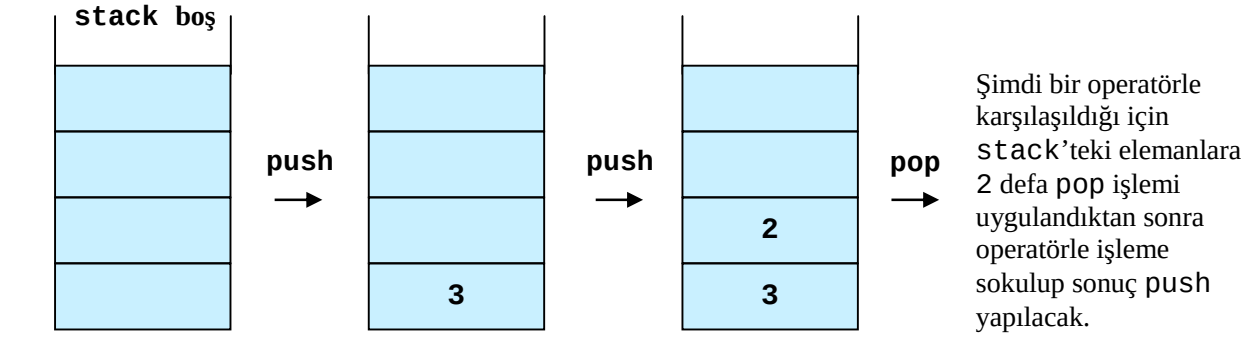
Dikkat edilecek olunursa, postfix ile prefix ifadeler birbirinin ayna görüntüsü değillerdir. Şimdi bir örnekle notasyon işlemlerinin `stack` kullanılarak nasıl gerçekleştirildiklerini görelim.

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

Çözüm algoritması şöyle olmalıdır;

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

Bu adımları uygulayarak örneğin çözümünü şekillerle açıklayalım.

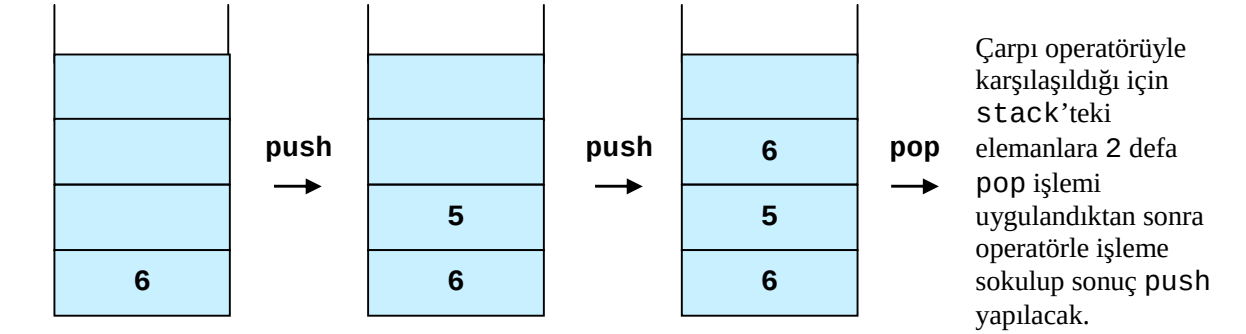


Boş bir `stack`. Bir operatörle karşılaşıncaya kadar `push` yapılacak.

`3` sayısı `stack`'e eklendi.

`2` sayısı `stack`'e eklendi.

Şimdi bir operatörle karşılaşıldığı için `stack`'teki elemanlara `2` defa `pop` işlemi uygulandıktan sonra operatörle işleme sokulup sonuç `push` yapılacak.

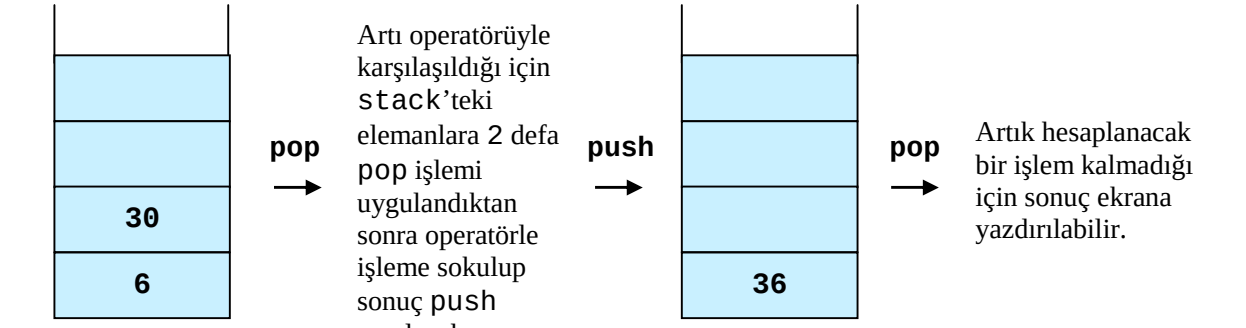


Çarpım sonucu olan 6 sayısı `stack`'e eklendi.

`5` sayısı `stack`'e eklendi.

`6` sayısı `stack`'e eklendi.

Çarpı operatörüyle karşılaşıldığı için `stack`'teki elemanlara `2` defa `pop` işlemi uygulandıktan sonra operatörle işleme sokulup sonuç `push` yapılacak.



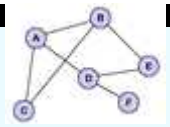
Çarpım sonucu olan 30 sayısı `stack`'e eklendi.

Artı operatörüyle karşılaşıldığı için `stack`'teki elemanlara `2` defa `pop` işlemi uygulandıktan sonra operatörle işleme sokulup sonuç `push` yapılacak.

`36` sayısı `stack`'e eklendi.

Artık hesaplanacak bir işlem kalmadığı için sonuç ekrana yazdırılabilir.

Şekil 3.9 Postfix ifadesinin `stack` kullanılarak hesaplanması.



BÖLÜM

Queues (*Kuyruklar*) 4

4.1 GİRİŞ

Kuyruk (*queue*) veri yapısı `stack` veri yapısına çok benzer. Kuyruk veri yapısında da veri ekleme (*enqueue*) ve kuyruktan veri çıkarma (*dequeue*) şeklinde iki tane işlem tanımlıdır. Yığınların tersine FIFO (*First In First Out*) prensibine göre çalışırlar ve ara elemanlara erişim doğrudan yapılamaz. Veri ekleme `stack`'teki gibi listenin sonuna eklenir. Fakat veri çıkarılacağı zaman listenin son elemanı değil ilk elemanı çıkarılır. Bu ise, kuyruk veri yapısının ilk giren ilk çıkar tarzı bir veri yapısı olduğu anlamına gelir.

Kuyruğun çalışma mantığını anlatan gündelik hayatta karşılaştığımız olaylara birkaç örnek verirsek;

- sinema gişesinden bilet almak için bekleyen insanlar,
- bankamatikten para çekmek için bekleyen insanlar,

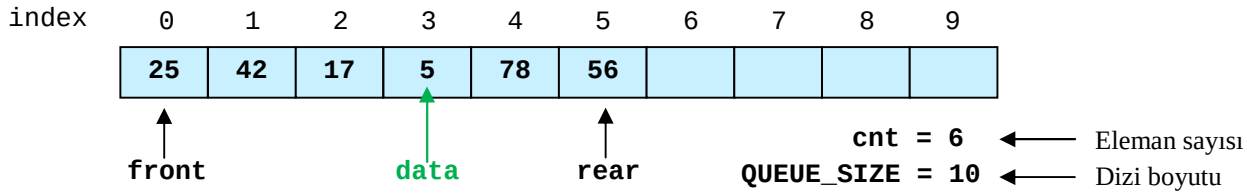
gibi, kuyruğa ilk gelen kişi ilk hizmeti alır (*ilk işlem görür*) ve kuyruktan çıkar.

Kuyruk (*queue*) yapısı bilgisayar alanında; ağ, işletim sistemleri, istatistiksel hesaplamalar, simülasyon ve çoklu ortam uygulamalarında yaygın olarak kullanılmaktadır. Örneğin, yazıcı kuyruk yapısıyla işlemleri gerçekleştirmektedir. Yazdır komutu ile verilen birden fazla belgeden ilki yazdırılmakta, diğerleri ise kuyrukta bekletilmekte, sırası geldiğinde ise yazdırılmaktadır.

Kuyruk veri yapısı bir sabit dizi olarak gösterilebilir veya bir bağlı liste olarak tanımlanabilir. Kuyrukların array implementasyonlarında, kuyruğun eleman sayısını tutan bir değişken ile kuyruğun başını gösteren (*genellikle* ***`front`*** *olarak isimlendirilir*) ve kuyruğun sonunu gösteren (*genellikle* ***`rear`*** *olarak isimlendirilir*) üç adet `int` türden değişken ile saklanacak verinin türüne uygun bir dizi bulunur. Kuyrukların bağlı liste implementasyonlarında ise kuyruğun başını ve sonunu gösteren `struct node` türden iki işaretçi ile eleman sayısını tutan `int` türden bir counter'ı bulunur. Kuyruğun başını gösteren işaretçi her veri çıkarıldığında bir sonraki veriyi gösterir. Kuyruğun sonunu gösteren işaretçi ise her veri eklendiğinde yeni gelen veriyi gösterecek şekilde değiştirilir.

Kuyruk (*queue*) yapısında da yığın yapısında olduğu gibi eleman ekleme ve eleman çıkarma olmak üzere iki işlem söz konusudur. Kuyruğa eleman eklemek için kuyruğun dolu olmaması gerekir ve ilk olarak kuyruğun dolu olup olmadığı kontrol edilir ve boş yer varsa eleman eklenir; boş yer yoksa ekleme işlemi başarısız olur. Kuyruğa eleman ekleme işlemi **ENQUEUE** olarak adlandırılır. Kuyruktan eleman çıkarmak için de kuyruğun boş olmaması gerekir. Bu çıkarma işlemi için de ilk olarak kuyruğun boş olup olmadığı kontrol edilir ve kuyruk boş değilse, eleman çıkarma işlemi gerçekleştirilir. Eğer boşsa eleman çıkarma işlemi başarısız olur. Kuyruktan eleman çıkarma işlemi **DEQUEUE** olarak bilinir.

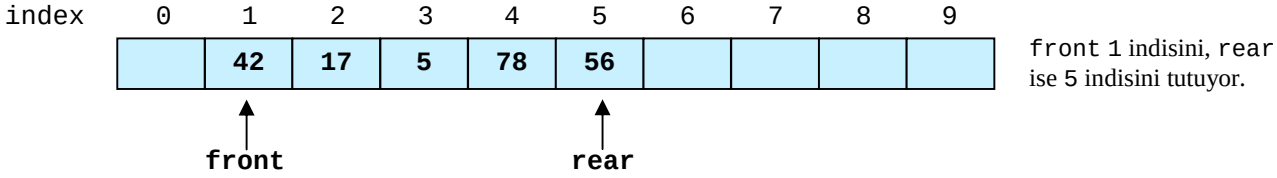
4.2 KUYRUKLARIN DİZİ (Array) İMPLEMENTASYONU



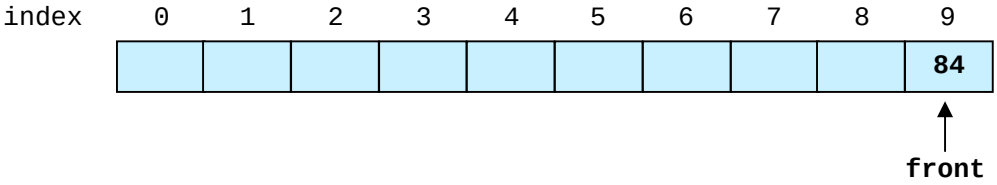
Şekil 4.1 Kuyruklarda (*queue*) array implementasyonunun mantıksal gösterimi.

Müşteri kuyruklarında öndeki müşteri beklediği hizmeti alıp kuyruktaki yerinden ayrılınca, kuyruktakiler birer adım kolayca, ön tarafa istekli biçimde yürürler. Yazılım içinde oluşturulmuş olan kuyruklarda ise bu işlem kendiliğinden ve kolay uygulanır bir nitelik değildir. Basit yapılı kuyruk denen bu modelin sakıncası, kuyruktaki öğelerin sayısı kadar kaydırma işleminin kuyruktan her ayrılma olayında yapılmasıdır. Ekleme ve çıkarma işlemlerinde ön indeks (*`front`*) daima sabittir. Çıkarma (*`dequeue`*) işleminde, kuyruğun önündeki eleman çıkarılır ve diğer bütün elemanlar bir öne kayar, ayrıca `rear` değişkeninin değeri de `1` azaltılır. Ekleme (*`enqueue`*) işleminde ise arka indeks (*`rear`*) dizide ileri doğru hareket eder. Şekil 4.1'de basit yapılı bir kuyruk modeli verilmiştir.

Sürekli kaydırma işleminin maliyeti düşünüldüğünde eleman çıkarmalarda verileri kaydırmak yerine `front` ön indeksi kaydırılsın şeklinde bir çözüm akla gelebilir. Fakat bu defa da başka bir problem ortaya çıkmaktadır. Yukarıdaki şekil dikkate alındığında kuyruklarda `enqueue` işlemi sağ taraftan, `dequeue` işlemi ise sol taraftan yapılmaktadır. Şekil 4.2'de görüldüğü gibi verileri kaydırmayıp `front` indeksini kaydıracağımıza göre kuyruğun solundan bir eleman çıkartıldığında orada boşluk oluşacak, `front` ise `1` no'lu indisi gösterecektir. Sürekli bir `dequeue` işleminde başka bir önlem alınmazsa `front` indeksi dizi sonuna kadar ulaşabilecektir. Bu durumu da Şekil 4.3'te görüyorsunuz.

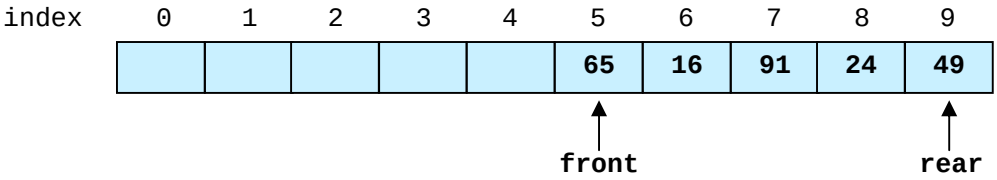


Şekil 4.2 `dequeue` işlemi sonrası kuyruğun görünümü.



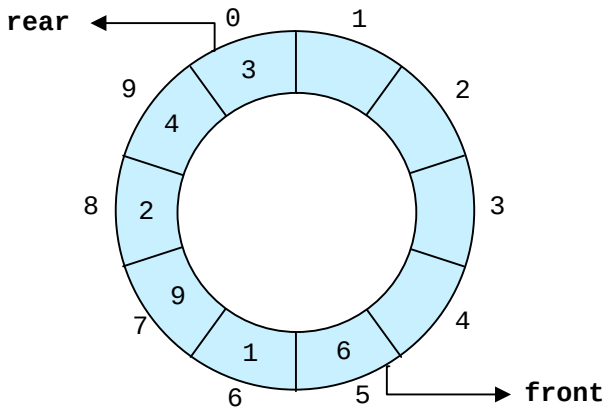
Şekil 4.3 `front` indeksi kuyruğun son indisini gösteriyor.

Şimdi Şekil 4.2'deki duruma göre birkaç defa `enqueue` ve `dequeue` işlemleri sonucunda Şekil 4.4'te görüldüğü gibi `rear` indeksinin kuyruğun son indisine kadar geldiğini kabul edelim. Eleman eklemek istediğimizde bu defa `overflow` hatasıyla karşılaşacağız demektir. Ayrıca kuyruğun solunda bulunan boşluklara da eleman eklenemeyecektir.



Şekil 4.4 Basit yapılı kuyrukta meydana gelen problem.

Böyle bir durumdan kurtulmanın yolu, kuyruğa dairesel dizi (*circular array*) efekti vermektir. Kuyruk işlemlerinde sürekli baştan ya da sondan çıkarma veya ekleme yaptığımıza göre elemanlar arasında boşluk yok demektir. Boşluklar ya baş taraftadır, ya da kuyruğun son tarafındadır. Kuyruklarda `enqueue` işleminde önce `rear` indeksi `1` arttırılır ve sonra eleman eklenir. `rear` kuyruğun son indisini gösterdiğinde ekleme yapmadan önce `1` arttırıldığına göre indeks dizi boyutuna eşit olacak ve taşma hatası meydana gelecektir. Bunun önüne geçmek için `rear` indeksi `0` olacak şekilde update edilerek elemanın kuyruğun `0` no'lu indisine eklenmesi sağlanır. Peki ama bir kuyrukta `0` no'lu indisin boş olduğunu nereden bileceğiz? Bu sorunun cevabı listeler ve yığınlarda olduğu gibi kuyruk dolu mu şeklinde bir kontrol yapmak olmalıdır. Eğer dolu değilse en azından `0` no'lu indisin boş olduğu kesin olarak bilinmektedir. Gerçekte dairesel bir dizi olmamasına karşın, böyle bir efekt verilmiş model Şekil 4.5'te gösterilmiştir.



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

Şekil 4.5'te kuyruğun sonundayken dairesel efekt vermek için `rear` indeksinin değeri `0` yapılıyor ve ilk indise eleman ekleniyor. Şimdi `rear 0`, `front` ise `5` indis değerini tutuyor. `rear` indeksinin `front` indeksinden daha önde yer

alması uygulanan efektin bir sonucudur. Artık kuyruk modeli biçimlendiğine göre array implementasyonlarındaki yapı tanımlanabilir. Önce kuyruk boyutunu belirleyen sabiti `define` bildirimiyle 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 Kuyruğa Başlangıç Değerlerini Vermek (`initialize`)

`queue` yapısını kullanmadan önce `initialize` etmek gerekir. Fonksiyon oldukça basittir ve yapı değişkenlerinin ilk değerlerini vermektedir.

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

Kuyruğun Boş Olup Olmadığının Kontrolü (isEmpty)

Yığınlarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı değişkeni `0` ise kuyruk boştur ve geriye `true` döndürülür. Eğer `cnt 0`'dan büyükse kuyrukta eleman vardır ve geriye `false` değeri döndürülür.

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

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

Kuyruğun Dolu Olup Olmadığının Kontrolü (isFull)

Yığınlarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı değişkeni `QUEUE_SIZE`'a eşit ise kuyruk doludur ve geriye `true` döndürülür. Eğer `cnt QUEUE_SIZE`'dan küçükse kuyrukta dolu değildir ve geriye `false` değeri döndürülür.

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

Kuyruğa Eleman Eklemek (enqueue)

Bir kuyruğa eleman ekleme fonksiyonu `enqueue` olarak bilinir ve bu fonksiyon, kuyruğun adresini ve eklenecek elemanın değerini parametre olarak alır. Geri dönüş değeri olmayacağı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için kuyruğun 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 değeri kontrol ediliyor. rear son
            indisi gösteriyor ve kuyruk dolu değilse rear değeri 0'a set ediliyor */
        q -> data[q -> rear] = x;
    }
}
```

Eleman eklendikten sonra `rear` değişkeni en son eklenen elemanın indis değerini tutmaktadır.

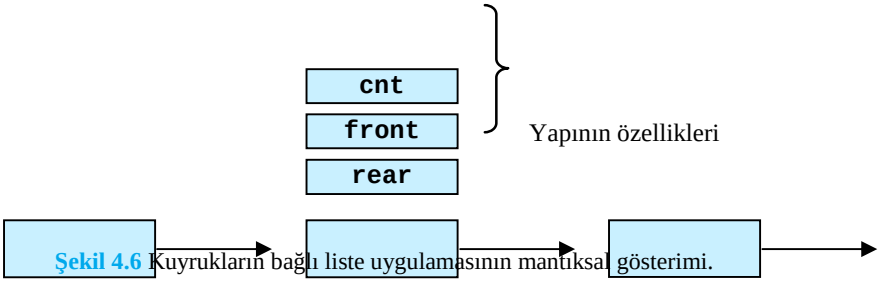
Kuyruktan Eleman Çıkarma İşlemi (dequeue)

`dequeue` fonksiyonu olarak bilinir. İlk giren elemanı çıkarmak demektir. Çıkarılan elemanın veri değeriyle geri döneceği için fonksiyonun tipi `int` olmalıdır.

```
int dequeue(queue *q) {
    if(!isEmpty) { // kuyruk boş mu diye kontrol ediliyor
        int x = q -> data[q -> front]; // front değerinin 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 değeriyle çağrıldığı yere geri dönüyor
    }
}
```

4.3 KUYRUKLARIN BAĞLI LİSTE (*Linked List*) İMPLEMENTASYONU

Kuyrukların bağlı liste uygulamasında, üç adet özellikleri bulunmaktadır. Bunlardan `cnt` eleman sayısını tutan `int` türden bir yapı değişkenidir. İkincisi `struct node` türden `front` işaretçisi kuyruğun önünü göstermektedir ve üçüncüsü ise yine `struct node` türden `rear` işaretçisidir. `rear` ise kuyruğun arkasını göstermektedir. Bu uygulamada eleman ekleme işlemi `rear` üzerinden, çıkarma işlemi ise `front` üzerinden yapılır. Bunlar bir kuyrukta ekleme ve çıkartma işlemlerini kolay bir şekilde yapabilmek için tanımlanmıştır. Yine ekleme ve çıkarma işlemlerinde kolay erişim için bir de `cnt` değişkeni tanımlanmıştır.



Şekil 4.6 Kuyrukların bağlı liste uygulamasının mantıksal gösterimi.

Veri yapısı aşağıdaki gibi tanımlanmıştır.

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

Kuyruğa Başlangıç Değerlerini Vermek (`initialize`)

Her zaman olduğu gibi ilk önce `initialize` yapılması gerekiyor. Fonksiyon şu şekilde tanımlanabilir;

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

`initialize` fonksiyonunda `q -> front = q -> rear = NULL;` satırındaki gibi bir işlemde `NULL` değeri ilk önce `q -> rear`'a atanır. Bu defa da `q -> front` göstericisine `q -> rear` değeri atanarak iki işaretçi de `NULL` değerini almış olur.

Kuyruğun Boş Olup Olmadığının Kontrolü (isEmpty)

Eleman sayısını tutan `cnt` yapı değişkeni `0` ise kuyruk boştur ve geriye `true` döndürülür. Eğer `cnt 0`'dan büyükse kuyrukta eleman vardır ve geriye `false` değeri döndürülür.

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

Kuyruğun Dolu Olup Olmadığının Kontrolü (`isFull`)

Yığınlarda ki fonksiyonla aynıdır. Eleman sayısını tutan `cnt` yapı değişkeni `QUEUE_SIZE`'a eşit ise kuyruk doludur ve geriye `true` döndürülür. Eğer `cnt` `QUEUE_SIZE`'dan küçükse kuyrukta dolu değildir ve geriye `false` değeri döndürülür.

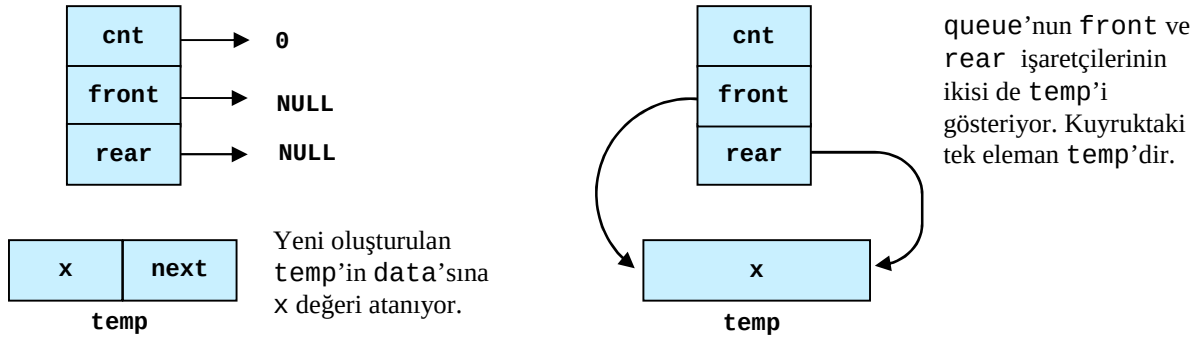
```
int isFull(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}
```

Kuyruğa Eleman Eklemek (enqueue)

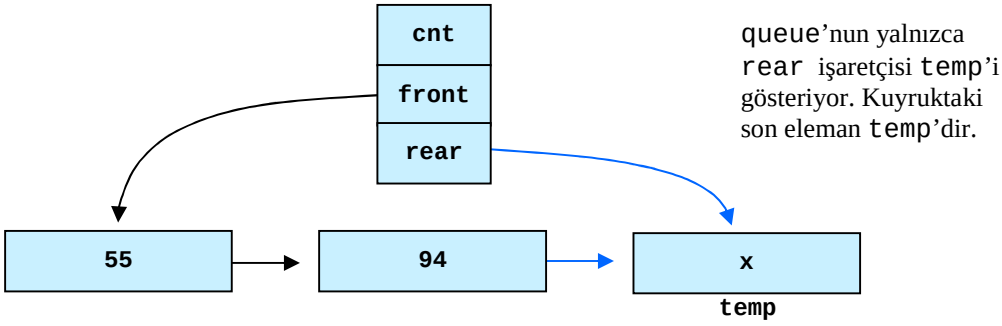
Bir kuyruğa eleman ekleme fonksiyonu `enqueue` olarak bilinir ve bu fonksiyon, kuyruğun adresini ve eklenecek elemanın değerini parametre olarak alır. Geri dönüş değeri olmayacağı için türü `void` olmalıdır. Ayrıca eleman ekleyebilmek için kuyruğun dolu olmaması gerekir. `stack`'lerdeki `push` işlemi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



Şekil 4.7 a) Boş bir kuyruğa eleman eklenmesi.



Şekil 4.7 b) Dolu bir kuyruğa eleman eklenmesi.

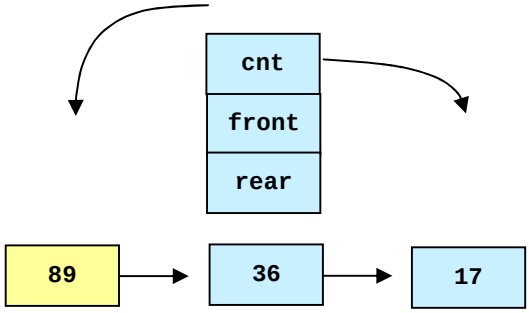
Ekleme yapmadan önce `rear` içinde `94` verisi bulunan düğümü gösteriyordu. Şimdi ise `temp`'i gösteriyor. Burada anlaşılması gereken nokta şudur; `queue` yalnızca elemanların başını, sonunu ve sayısını tutan tek bir yapıdır. Eklenen tüm veriler `struct` node türden birer düğümdür.

Kuyruktan Eleman Çıkarma İşlemi (dequeue)

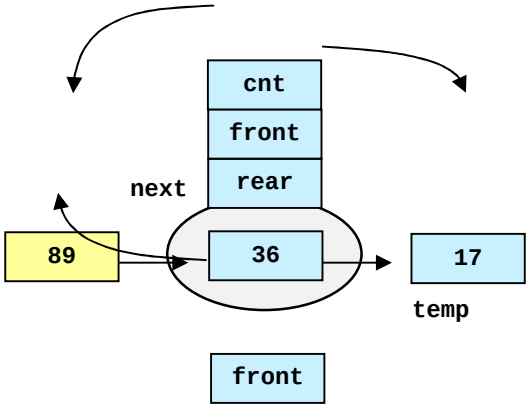
Bir kuyruktan eleman çıkarma fonksiyonu `dequeue` olarak bilinir ve fonksiyon, kuyruğun adresini parametre olarak alır. İlk giren elemanı çıkarmak demektir. Çıkarılan elemanın veri değeriyle geri döneceği için fonksiyonun tipi `int` olmalıdır. Eleman çıkartabilmek için kuyruğun boş olmaması gerekir. `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 değeriyle çağrıldığı yere geri dönüyor
    }
}
```

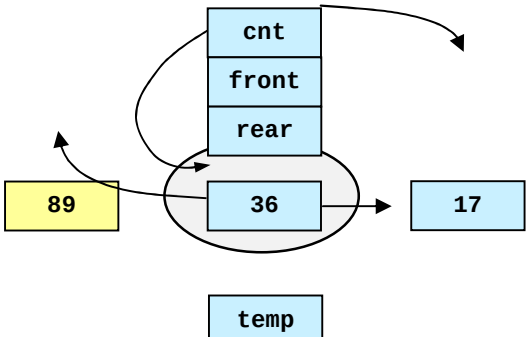
Kuyruğun `front` işaretçisinin gösterdiği adres `temp`'e atanmıştır. Şimdi aynı düğümün adresi hem `front` işaretçisinde hem de `temp` işaretçisindedir. Şekilde `dequeue` işlemi açık bir şekilde olarak gösterilmiştir.



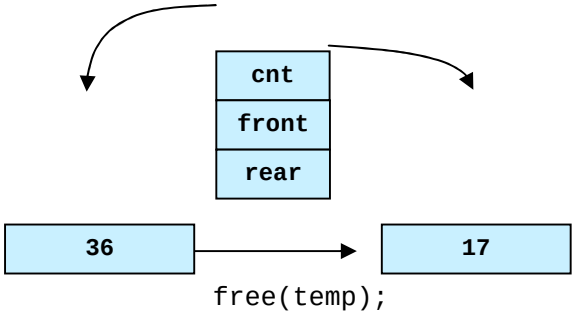
Şekil 4.8 a) Kuyruktan çıkarılacak eleman sarı renkli olarak görülüyor.



Şekil 4.8 b) Şimdi `temp` de silinecek elemanı gösteriyor.



Şekil 4.8 c) `front` bir sonraki elemanı gösteriyor.



Şekil 4.8 d) Kuyruktan eleman silme işlemi tamamlanıyor.

Silinecek düğümün `next` işaretçisi bir sonraki düğümün adresini tutmaktadır (*`front->next`, ve `temp->next` de aynı adresi tutuyor demektir*). Şu anda hem `front`'un hem de `temp`'in `data`'sı olan `89` değeri `x` değişkenine aktarılıyor. Sonra `temp->next` değeri `front` işaretçisine atanarak `36` verisinin bulunduğu düğümü göstermesi sağlanıyor. Nihayet `temp` işaretçisinin gösterdiği adres `free()` fonksiyonuyla belleğe geri kazandırılıyor ve eleman sayısını tutan `cnt` değişkeni de `1` azaltılarak `dequeue` işlemi tamamlanıyor.

Örnek 4.1: Palindrom, tersten okunuşu 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 edildiğini 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);
}
```

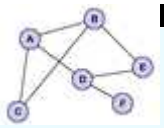
```
int isFull_queue(queue *q) {
    return(q -> cnt == QUEUE_SIZE);
}

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;
}
```



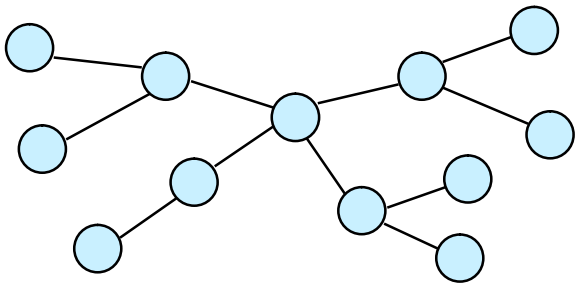
BÖLÜM

Ağaçlar (*Trees*) 5

5.1 GİRİŞ

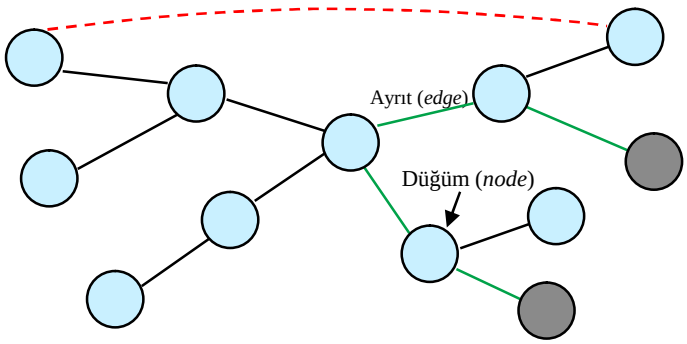
Ağaç, verilerin birbirine sanki bir ağaç yapısı oluşturuyormuş gibi sanal olarak bağlanmasıyla elde edilen hiyerarşik yapıya sahip bir veri modelidir; bilgisayar yazılım dünyasında, birçok yerde / uygulamada programcının karşısına çıkar. Ağaç veri yapılarının işletim sistemlerinin dosya sisteminde, oyunların olası hamlelerinde ve şirketlerdeki organizasyon şeması 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 ağaç veri yapısı kullanılır. Bunun sebebi astlık üstlük ilişkileridir. Arama algoritmalarında *(ikili arama – binary search*) kullanılabilmektedir. Daha değişik bir örnek verecek olursak, bir satranç tahtası üzerinde atın 64 hamlede tüm tahtayı dolaşması problemi ağaç veri yapısıyla çözülebilen bir problemdir. Birçok problemin çözümü veya modellenmesi, doğası gereği ağaç veri modeline çok uygun düşmektedir. Ağaç veri modeli daha fazla bellek alanına gereksinim duyar. Çünkü ağaç veri modelini kurmak için birden çok işaretçi değişken kullanılır. Buna karşın, yürütme zamanında sağladığı getiri ve ağaç üzerinde işlem yapacak fonksiyonların rekürsif yapıda kolayca tasarlanması ve kodlanması ağaç veri modelini uygulamada ciddi bir seçim yapmaktadır.

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



Şekil 5.1 Bir ağaç yapısının görünüşü.

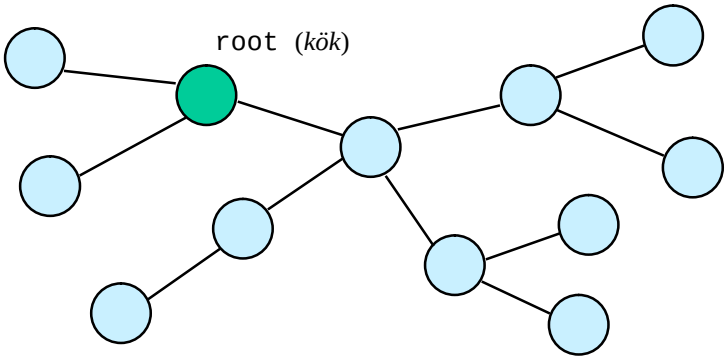
Şekil 5.1'de görülen yapı bir ağaçtır ve mutlaka bir ağaçta kök bulunmak zorunda değildir. Herhangi iki düğüm arasında yalnızca bir yol vardır ve bir çevrim içermemektedir.



Şekil 5.2 Bir ağaçta herhangi iki düğüm arasında yalnızca bir yol bulunur (*yeşil ayrıtların olduğu yol*).

Şekil 5.2'de koyu gri renkte gösterilmiş olan düğümler arasında yalnızca bir yol bulunmaktadır. Eğer kesikli kırmızı çizgi ile belirtilen bir çevrim (*cycle*) daha olsaydı, bu bir ağaç değil, sadece graf olacaktı. Çünkü düğümlere ikinci bir yol ile de erişebilme imkânı ortaya çıkmaktadır.

Kökü Olan Ağaç (*Rooted Tree*): Kökü olan ağaçta özel olan ya da farklı olan düğüm kök (*root*) olarak adlandırılır. Kök hariç her **c** (*child*) düğümünün bir **p** ebeveyni (*parent*) vardır. Alışılagelmiş şekliyle ağaçlarda, kök en yukarıda yer alıp çocuklar alt tarafta olacak biçimde çizilir. Fakat böyle gösterme zorunluluğu yoktur.



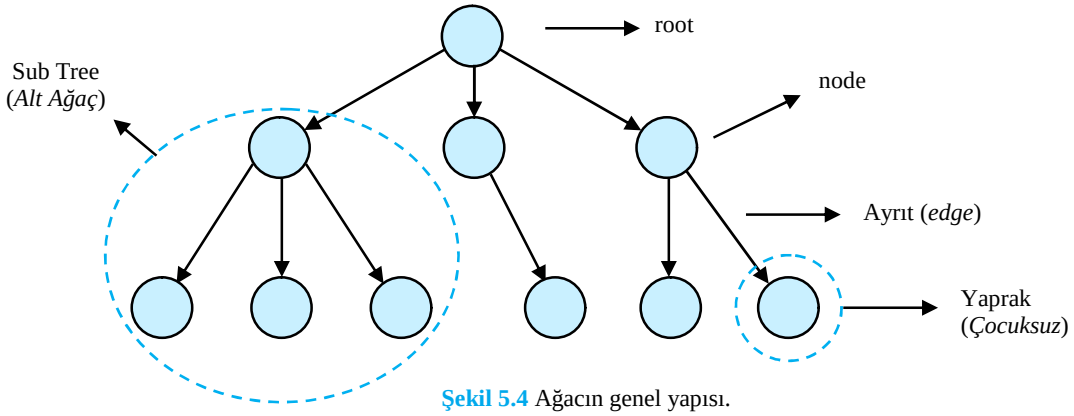
Şekil 5.3 Rooted tree modeli.

Ebeveyn (*Parent*): Bir **c** düğümünden köke olan yol üzerindeki ilk düğümdür. Bu **c** düğümü **p**'nin çocuğudur (*child*).
Yaprak (*Leaf*): Çocuğu olmayan düğümdür.
Kardeş (*Sibling*): Ebeveyni aynı olan düğümlerdir.
Ata (*Ancestor*): Bir d düğümünün ataları, d'den köke olan yol üzerindeki tüm düğümlerdir.
Descendant: Bir düğümün çocukları, torunları vs. gibi sonraki neslidir.
Yol Uzunluğu: Yol üzerindeki ayrıt sayısıdır.
n Düğümünün Derinliği: Bir **n** düğümünden köke olan yolun uzunluğudur. Kökün derinliği sıfırdır.
n Düğümünün Yüksekliği: Bir **n** düğümünden en alttaki descendant düğümüne olan yolun uzunluğudur. Başka bir deyişle neslinin en sonu olan düğümdür.
Bir Ağacın Yüksekliği: Kökün yüksekliğine ağacın yüksekliği de denir.
n Düğümünde Köklenen Alt Ağaç (*Subtree Rooted at n*): Bir n düğümü ve n düğümünün soyu (*descendant*) tarafından oluşan ağaçtır.
Binary Tree (*İkili Ağaç*): İkili ağaçta bir düğümün sol çocuk ve sağ çocuk olmak üzere en fazla iki çocuğu olabilir.
Düzey (*level*) / Derinlik (*depth*): Kök ile düğüm arasındaki yolun üzerinde bulunan ayrıtların sayısıdır. Bir düğümün kök düğümden olan uzaklığıdır. Kökün düzeyi sıfırdır (*bazı yayınlarda 1 olarak da belirtilmektedir*).

5.2 AĞAÇLARIN TEMSİLİ

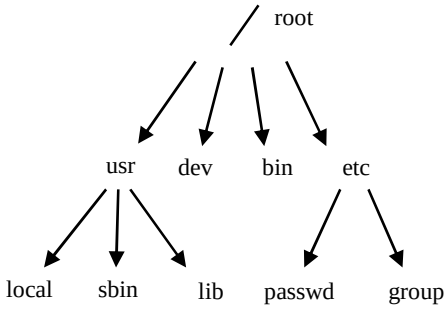
Genel ağaç yapısında düğümlerdeki çocuk sayısında ve ağaç yapısında bir kısıtlama yoktur. Ağaç yapısına belli kısıtlamalar getirilmesiyle ağaç türleri meydana gelmiştir. Her yaprağın derinliği arasındaki fark belli bir sayıdan fazla (*örneğin 2*) olmayan ağaçlara **dengeli ağaçlar** (*balanced trees*) denir. İkili ağaçlar (*binary trees*) ise düğümlerinde en fazla iki bağ içeren (0, 1 veya 2) ağaçlardır. Ağaç yapısı kısıtlamaların az olduğu ve problemin kolaylıkla uyarlanabileceği bir yapı olduğundan birçok alanda kullanılmaktadır. Örnek olarak işletim sistemlerinde kullandığımız dosya-dizin yapısı tipik bir ağaç modellemesidir.

Ağaç veri modelinde, bir kök işaretçisi, sonlu sayıda düğümleri ve onları birbirine bağlayan dalları vardır. Veri ağacın düğümlerinde tutulur. Dallarda ise geçiş koşulları vardır. Her ağacın bir kök işaretçisi bulunmaktadır. Ağaca henüz bir düğüm eklenmemiş ise ağaç boştur ve kök işaretçisi `NULL` değerini gösterir. Ağaç bu kök etrafında dallanır ve genişler.



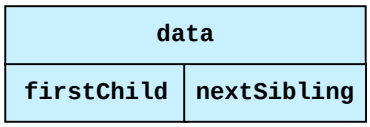
Şekil 5.4 Ağacın genel yapısı.

1) Tüm Ağaçlar İçin: 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 dönmüş bir ağaç gibidir.



Şekil 5.5 UNIX dosyalama hiyerarşisi modeli.

Şekil 5.5'de gösterilen dosya hiyerarşisindeki ağaç ikili bir ağaç değildir. Çünkü bazı düğümlerin üç çocuğu, kökün ise dört çocuğu bulunmaktadır. Binary ağaçlarda bir sol çocuk, bir de sağ çocuk bulunmaktaydı. Oysa bu ağaç türünde diğer çocuklar da yer alıyor. Şekil 5.5'de görüldüğü gibi `root`'un çocukları olan `usr`'yi `left` ile gösterdiğimizi, `dev`'i de `right` ile gösterdiğimizi düşünelim. Peki `bin` ile `etc`'yi nasıl gösterebiliriz? Bu durumda farklı bir veri yapısı tanımlamamız gerekecektir. Bu yapının şekli aşağıda gösterilmiştir.

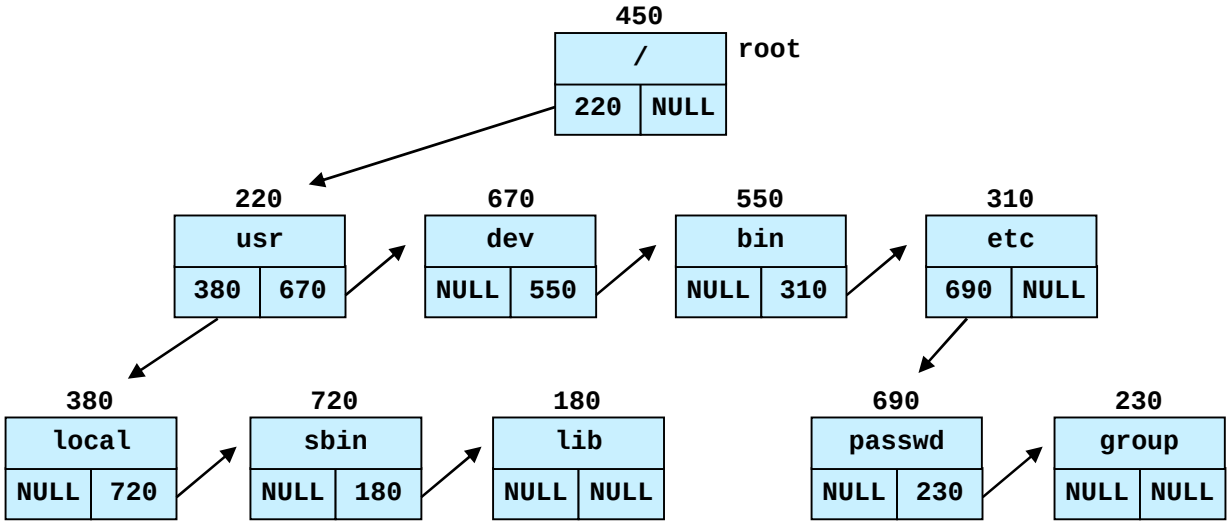


Şekil 5.6 İkili olmayan ağaçlarda bir `node`'un yapısı.

Kök'ün kardeşi yoktur, sadece çocukları ve torunları olabilir. İkili olmayan ağaçların gösteriminde her ebeveyn yalnızca ilk çocuğunu göstermeli ve eğer varsa kendi kardeşini de göstermelidir. Kardeşler birden fazla olabilir. Öyleyse ilk çocuktan sonra kardeşler birbirlerine bağlanmalıdır. Bu açıklamalara uygun veri yapısı aşağıdaki 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 olamayacağı için yalnızca `firstChild` bağlantısı var gibi görülse de diğer kardeşlerin düğümüne `firstChild` işaretçisinin gösterdiği ilk çocuk düğümünün `nextSibling` işaretçisinden erişilebilir. **Bu yöntem bellek alanından kazanç sağlar;** ancak, programın yürütme zamanını arttırır. Çünkü bir düğümden onun çocuklarına doğrudan erişim ortadan kalkmış, bağlı listelerde olduğu gibi ardışıl erişme ortaya çıkmıştır. Aşağıda, Şekil 5.5'deki UNIX dosya yapısının mantıksal modellemesi gösterilmiştir.



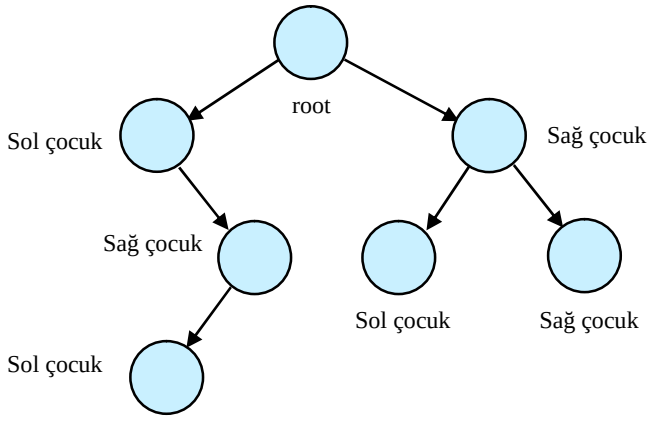
Şekil 5.7 UNIX işletim sistemindeki dosya yapısının veri modeli.

2) İkili Ağaçlar (*Binary Trees*) İçin: Eğer bir ağaçtaki her düğümün en fazla 2 çocuğu varsa bu ağaca ikili ağaç denir. Diğer bir deyişle bir ağaçtaki her düğümün derecesi en fazla 3 ise o ağaca ikili ağaç denir. İkili ağaçların önemli özelliği eğer dengeli ise çok hızlı bir şekilde ekleme, silme ve arama işlemlerini yapabilmesidir. Ağaç veri yapısı üzerinde işlem yapan fonksiyonların birçoğu kendi kendini çağıran (*recursive*) fonksiyonlardır. Çünkü bir ağacı alt ağaçlara böldüğümüzde elde edilen her parça yine bir ağaç olacaktır ve fonksiyonumuz yine bu ağaç üzerinde işlem yapacağı için kendi kendini çağıracaktır.

İkili ağaç veri yapısının tanımında ağaç veri yapısının tüm tanımları geçerlidir, farklı olan iki nokta ise şunlardır:

i. Ağacın derecesi (*degree of tree*) 2'dir.

ii. Sol ve sağ alt ağacın yer değiştirmesiyle çizilen ağaç aynı ağaç değildir.



Şekil 5.8 İkili ağaçların gösterimi.

İkili ağaç (*binary tree*) veri yapısının yoğun biçimde kullanılır oluşu iki önemli uygulama alanı ile ilişkilidir. Sıralama algoritmaları ile derleyicilerde sözdizim çözümleme (*syntax analysis*), kod geliştirme (*code optimization*) ve amaç kod üretme (*object code generation*) algoritmaları bu veri yapısını kullanırlar. Ayrıca kodlama kuramı (*coding theory*), turnuva düzenleme, aile ağacı gösterimi ve benzerleri tekil kullanımlar da söz konusudur.

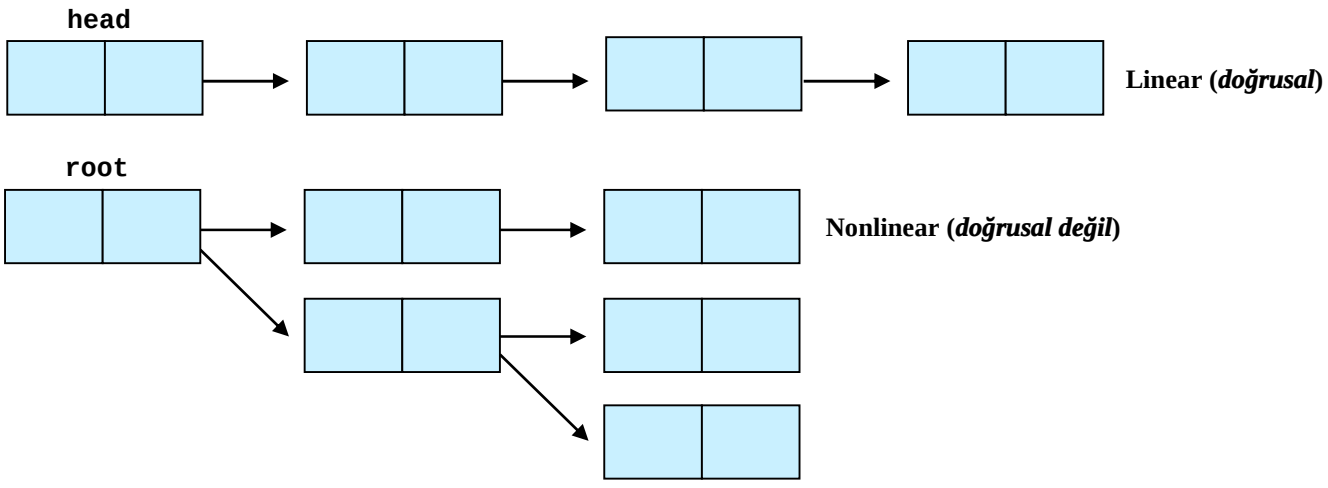
Ağaçlarla ilgili özellikler:

- Bir binary ağaçta, bir i seviyesindeki maksimum `node` sayısı 2^i 'dir. (i, 0 olduğu için).
- K derinliğine sahip bir binary ağaçta maksimum `node` sayısı $2^{K+1} - 1$ 'dir.
- Seviye ve derinlik kökle başlayacak şekilde aynı sayılmaktadır.
- Ağaçta `node`'ların eksik kısımları `NULL` olarak adlandırılır.

İkili bir ağaç için veri yapısı aşağıdaki gibi tanımlanabilir;

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

İhtiyaç olduğunda yapı içerisinde bir de `parent` tanımlanabilir. Ağaçların yapısının da bağlı liste olduğu tanımdan anlaşılabilir. Fakat normal bağlı listelerden farkı **nonlinear** olmasıdır. Yani doğrusal değildir.



Şekil 5.9 Ağaçlarla bağlı listelerin karşılaştırılması.

İkili Ağaçlar Üzerinde Dolaşma

İkili ağaç üzerinde dolaşma birçok şekilde yapılabilir Ancak, rastgele dolaşmak yerine, önceden belirlenmiş bir yönteme, bir kurala uyulması algoritmik ifadeyi kolaylaştırır. Üstelik rekürsif fonksiyon yapısı kullanılırsa ağaç üzerinde işlem yapan algoritmaların tasarımı kolaylaşır. Önce-kök (*preorder*), kök-ortada (*inorder*), sonra-kök (*postorder*) olarak adlandırılan üç değişik dolaşma şekli çeşitli uygulamalara çözüm olmaktadır.

1- Preorder (*Önce Kök*) Dolaşma: Önce kök yaklaşımında ilk olarak `root(kök)`, sonra `left` (*sol alt ağaç*) ve ardından `right` (*sağ alt ağaç*) dolaşılır. Fonksiyonu tanımlamadan önce kolayca kullanabilmek için `typedef` bildirimiyle bir yapı nesnesi oluşturulabilir. Bildirim `typedef struct node *BTREE;` şeklinde asterisk `(*)` konularak da yapılabilir. Böylece her `BTREE` türünde geriye adres döndüren fonksiyonların ve adres değişkenlerinin bildiriminde asterisk işaretinden kurtulunmuş olunur fakat bazı derleyicilerin kararsız çalışmasına neden olabilmektedir.

```
typedef struct node BTREE;
```

`BTREE` yapısı `struct node` veri yapısıyla tanımlanmış olan bir çeşit `node`'dur. İçerisinde veri tutan `data` değişkeni, adres bilgisi tutan `left` ve `right` isimli iki işaretçisi bulunmaktadır. İkili ağaçta dolaşırken fonksiyon geriye herhangi bir şey döndürmeyeceği ve sadece ağaç üzerinde dolaşacağı için türü `void` olmalıdır.

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

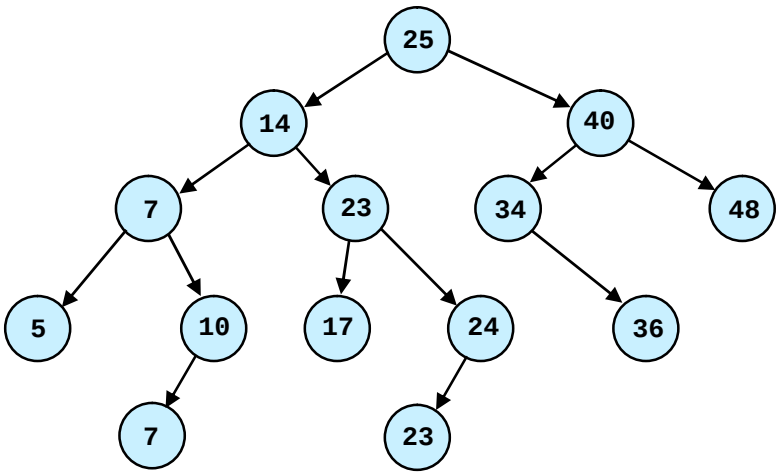
2- Inorder (*Kök Ortada*) Dolaşma: `Inorder` dolaşmada önce `left` (*sol alt ağaç*), sonra `root` (*kök*) ve `right` (*sağ alt ağaç*) dolaşılır. Fonksiyonun türü yine `void` olmalıdır.

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

3- Postorder (*Kök Sonda*) Dolaşma: `Postorder` yaklaşımında ise, önce `left` (*sol alt ağaç*), sonra `right` (*sağ alt ağaç*) ve `root` (*kök*) dolaşılır. Fonksiyon `void` türdendir.

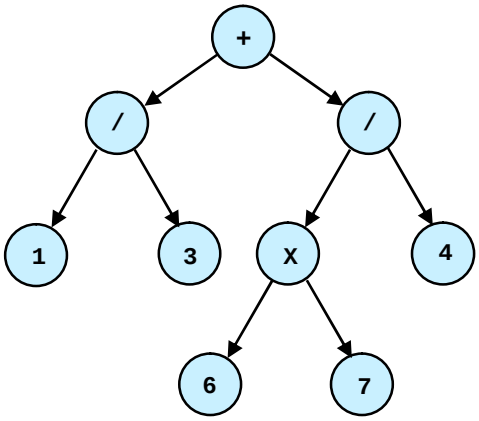
```
void postorder(BTREE *root) {
    if(root != NULL) {
        postorder(root -> left);
        postorder(root -> right);
        printf("%d", root -> data);
    }
}
```

Örnek 5.1: "25, 14, 23, 40, 24, 23, 48, 7, 5, 34, 10, 7, 17, 36" değerlerine sahip düğümler için ikili ağaç gösterimini oluşturunuz 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: Aşağıda görülen bağıntı ağacındaki 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 / +

İkili Ağaç Oluşturmak

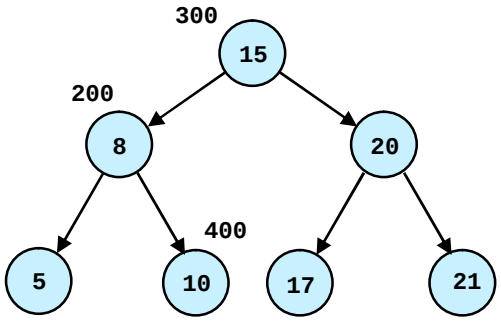
```
typedef struct node BTREE;

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;
}
```

İkili Ağaca Veri Eklemek

İkili bir ağaca veri ekleme işleminde sol çocuğun verisi (*data*) ebeveyninin verisinden küçük olmalı, sağ çocuğun verisi ise ebeveyninin verisinden büyük ya da eşit olmalıdır. Altta `insert` isimli ekleme fonksiyonunun nasıl yazıldığını görüyorsunuz. Fonksiyon, veri eklendikten sonra ağacın kök adresiyle geri dönmektedir.

```
/* İkili ağaca veri ekleyen fonksiyon */
BTREE *insert(BTREE *root, int data) {
// Fonksiyona gönderilen adresteki ağaca ekleme yapılacak
    if(root != NULL) { // ağaç boş değilse
        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 // eğer ağaç boş ise
        root = new_node(data);

    return root;
}
```

Fonksiyonun nasıl çalıştığını bir şekille açıklayalım. Şekil 5.10'da görülen ağaçta bazı düğümlerin adresleri üzerlerinde belirtilmiştir.



Şekil 5.10: Veri eklenecek örnek bir ağaç.

Ağaca 13 sayısının eklenecek olduğunu kabul edelim. Fonksiyon **`insert(300, 13);`** kodu ile çağrılacaktır.

```
if(root != NULL) { // 300 adresinde bir ağaç var, yani NULL değil ve koşul doğru
    if(data < root -> data) // 13 root'un data'sı olan 15'ten küçük ve koşul doğru
        root -> left = insert(root -> left, data); // yürütülecek satır
    else
        root -> right = insert(root -> right, data);
}
else
    root = new_node(data);
return root;
```

13, kökün değeri olan 15'ten küçük olduğu için sol çocuk olan 8 verisinin bulunduğu 200 adresiyle tekrar çağrılıyor.

```
insert(200, 13);
```

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

Bu defa 13, 8'den büyük olduğundan fonksiyon 8'in sağ çocuğu olan 10 verisinin bulunduğu 400 adresiyle çağrılıyor.

```
insert(400, 13);
```

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

Şimdi de fonksiyon 10'un sağ çocuğu ile tekrar çağrılıyor. Fakat `root->right` (*sağ çocuk*) düğümü henüz olmadığı için adres ile değil NULL ile fonksiyon çağrılacaktır.

```
insert(NULL, 13);
```

```
else // eğer ağaç boş ise yani düğüm yok ise
    root = new_node(data); // düğüm oluşturuluyor ve 13 ekleniyor
return root; // oluşturulan ve veri eklenen düğümün adresi geri döndürülüyor
```

400 adresindeki 10 datasını bulunduran düğümün sağ çocuğu olan right, NULL değere sahiptir. Yani herhangi bir düğümü göstermemektedir, çocukları yoktur. Fonksiyonun en sonunda geri döndürülen yeni düğümün adresi, 10'un sağ çocuğunu gösterecek olan ve NULL değere sahip `right` işaretçisine atanıyor. Sonra fonksiyon kendisinden önceki çağrıldığı noktaya geri dönüyor. Bölüm 3'te `stack` konusu ve rekürsif fonksiyonların çalışma prensibi anlatılmıştı. Fonksiyon, her geri dönüşte işleyiş 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: İkili 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 İKİLİ ARAMA AĞAÇLARI (BSTs - *Binary Search Trees*)

İkili ağaç yapısının önemli bir özelliği de düğümlerin yerleştirilmesi sırasında uygulanan işlemlerdir. İkili 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. İkili arama ağaçlarındaki **(BST - *Binary Search Tree*)** durum ise farklıdır. Bir ikili arama ağacındaki her düğüm, sol alt ağacındaki tüm değerlerden büyük, sağ alt ağacındaki tüm değerlerden küçük ya da eşittir. İkili arama ağaçlarında her bir düğümün alt ağacı yine bir ikili arama ağacıdır. inorder sıralama yapıldığında küçükten büyüğe veriler elde edilir.

İkili arama ağaçlarında (*BST*) aynı değerlere 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 düğümün 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 değerler eklenmelidir. Burada ise şöyle bir sorun ortaya çıkmaktadır. Tekrarlı değerlere sahip bir `BST` ağacından tekrarı olan bir değeri silmek istediğimizde hangisini sileceğiz? Ya da arama yapmak istediğimizde hangisini bulacağız?

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

Biz bu derste aşağıdaki algoritmada gösterildiği üzere eşit değerleri ağacın sağ tarafına ekleyeceğiz.

İkili Arama Ağacına Veri Eklemek

İkili 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.

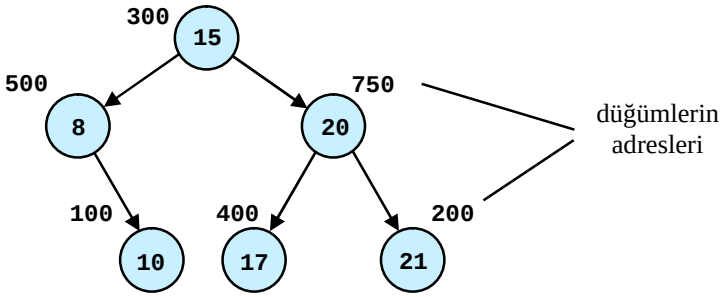
```
BTREE* insertBST(BTREE *root, int data) {
    if(root != NULL){
        if(data < root -> data) // data, root'un datasından küçükse
            root -> left = insert(root -> left, data);
        else // data, root'un datasından büyük ya da eşitse
            root -> right = insert(root -> right,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

Bir Ağacın Düğümlerinin Sayısını Bulmak

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

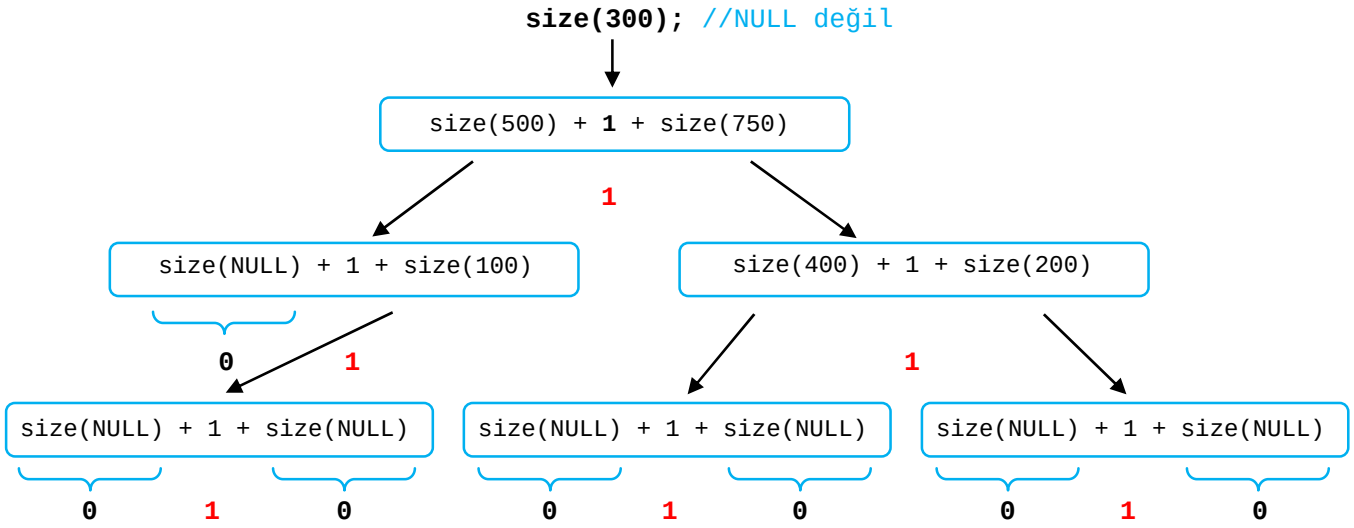
```
/* İkili bir ağacın düğüm sayılarını veren fonksiyon */
int size(BTREE *root) {
    if(root == NULL)
        return 0;
    else
        return size(root -> left) + 1 + size(root -> right);
}
```

Fonksiyonun nasıl çalıştığını inceleyelim.



Şekil 5.11 6 düğümü olan ikili bir arama ağacı.

Aşağıda Şekil 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` dolaşılıyor. 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` dolaşılıyor. **LIFO** (*Last in First Out*) **son giren ilk çıkar** prensibine göre `stack`'te tutulan rakamlar çıkarılacak ve toplanarak ağacın düğüm sayısı bulunacaktır.



Şekil 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 çağırmaktadır.

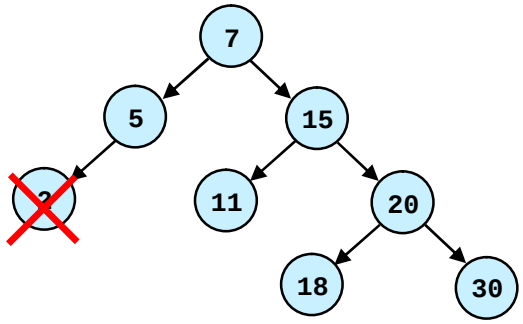
```
int height(BTREE *root) {
    if(root == NULL)
        return -1;
    else {
        int left_height, right_height;
        left_height = height(root -> left);
        right_height = height(root -> right);
        if(right_height > left_height)
            return right_height + 1;
        else
 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

İkili Arama Ağacından Bir Düğüm Silmek

Bir düğümü silmek istediğimizde ilk olarak fonksiyona parametre olarak gönderdiğimiz ağaçta eleman ya vardır veya yoktur. Eğer ağaçta hiç eleman yoksa fonksiyon `NULL` ile geri dönmelidir. Ağaçta eleman var ise, silme işlemi ancak aranılan değere bir eşitlik varsa gerçekleşecektir. Yani bir ağaçtan `8`'i silmek istiyorsak, ancak o ağaçta `8` içeren bir düğüm varsa silme işlemi gerçekleşir. Fakat silinecek düğüm ağacın çocuklarından herhangi biri olabilir. Bu durumda silinecek düğüm, üzerinde bulunduğumuz düğümden küçükse, sol tarafa doğru bir düğüm ilerleyip yeni bir kontrol yapmamız gerekir. Büyükse bu defa sağ tarafta bir düğüm ilerleyip kontrolü tekrarlamamız gerekmektedir. Aranılan düğüm bulunduğunda da silme işleminden önce üç durumdan bahsetmeliyiz. İkili bir arama ağacında bir düğümün en fazla iki çocuğu olabileceğini düşündüğümüzde bunlar;

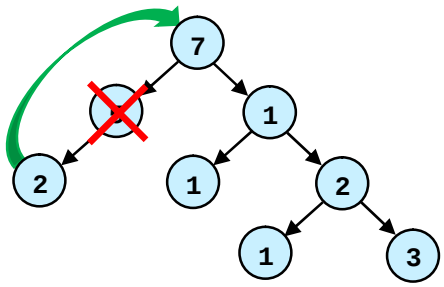
1- Silinecek olan düğümün çocuğu yok ise,
2- Silinecek olan düğümün sadece bir çocuğu var ise,
3- Silinecek olan düğümün iki çocuğu var ise,

nasıl bir algoritma kurmamız gerektiğidir. Bir örnek ile gösterelim;



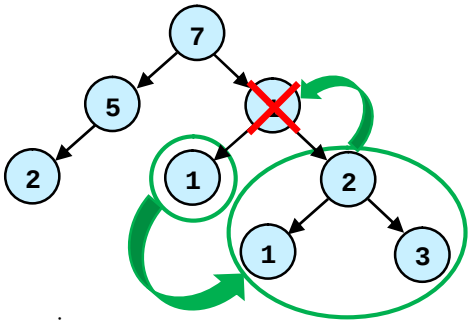
Şekil 5.13 İkili arama ağacından bir yaprağın silinmesi.

Diyelim ki silinecek olan veriler yapraklardan (`leaf`) biri olsun. Yani `2`, `11`, `18` ya da `30`'dan biri olsun. Bu durumda silinecek düğümü direk `free()` fonksiyonuyla yok edebiliriz. Yani `1` no'lu seçeneğin işi oldukça kolay. Bir çocuğu olan düğümü nasıl silebiliriz? Örneğin aşağıdaki şekilde görüldüğü gibi `5`'i silmek isteyelim. Bu durumda düğümün çocuğunu ebeveynine bağlar ve istenen düğümü silebiliriz. `2.`seçenek de kolay bir işlem gibi görünüyor.



Şekil 5.14 İkili arama ağacından bir çocuğu olan düğümün silinmesi.

Peki, Şekil 5.15'teki gibi `15`'i silmek istersek ne olacak? Bir düğümün kendisinin varsa sağ tarafındaki tüm çocuklardan küçük ya da eşit olduğu bilindiğine göre, sağ çocuklarından kendisine en yakın değerdeki düğüm, sağ tarafındaki çocuklarının en küçük düğümüdür. Öyleyse silinecek düğümün sol çocukları, sağ tarafın en küçük çocuğunun sol çocukları olacak şekilde bağlanır ve bu alt ağaç silinecek düğümün yerine kaydırılır.



Şekil 5.15 İkili arama ağacından iki çocuğu olan düğümün silinmesi.

Bu algoritmaya göre silme işlemi yapacak fonksiyonu tasarlayabiliriz. Fonksiyon verilen düğümü sildikten sonra ağacı tekrar düzenleyip kökle geri döneceğinden, geri dönüş değeri `root` yani kök olmalı ve türü de `BTREE` olmalıdır. Parametre olarak hangi ağaçtan silme işlemi yapılacaksa o ağacın kökünü ve silinecek veriyi (`x` *verisine sahip olan düğüm*) almalıdır. Şimdi `delete_node` isimli fonksiyonu yazalım.

```
BTREE *delete_node(BTREE *root, int x) {
    BTREE *p, *q;
    if(root == NULL)
        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;
}
```

Ağaç yoksa çalışacak olan kısım

1.durum

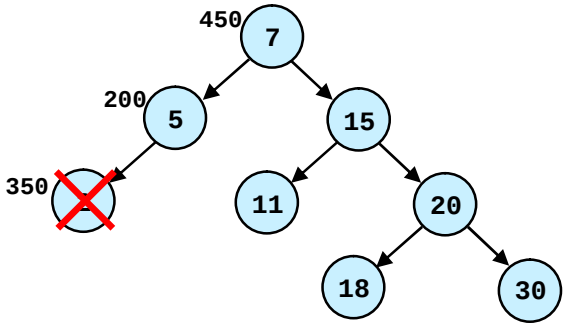
2.durum

3.durum

Aranan düğüm bulunmuşsa çalışacak olan kısım

Aranan düğüm henüz bulunamamışsa çalışacak olan kısım

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



Şekil 5.16 İkili arama ağacında birinci durum görünümü.

Şekilde düğümün adresinin 350 olduğu, kökün adresinin de 450 olduğu görülüyor. Kodu çalıştırdığımızda ilk olarak,

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

1

çağrısı yapılacak ve aranan düğüm henüz bulunamadığı 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

çağrısı yapılıyor. Dikkat edilirse fonksiyonun geri dönüş değeri `root->left`'e atanacak. Aranan düğüm yine bulunamadığı 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
```

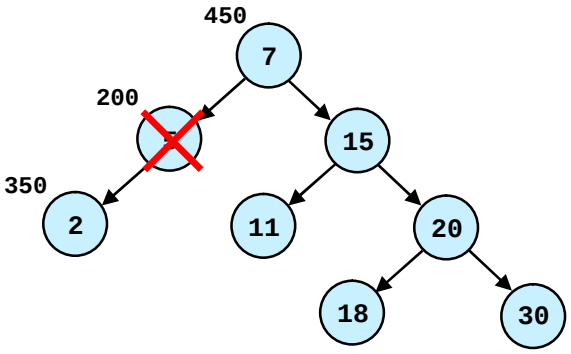
Şimdi `200` adresinin `left`'i `350` adresini gösteriyor;


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

Aranan düğüm şimdi bulunuyor. Düğüm bellekten siliniyor ve `root->left`'e `NULL` değeri 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 düğüm bellekten silindi
        return NULL; // Fonksiyon NULL döndürerek sonlandı.
    }
    ...
    return root;
}
```

Fonksiyon bir önceki çağrıldığı 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 çağrıldığı bir önceki noktaya yani `1` numaralı bölüme dönerek sonraki kod olan `return root` icra edilerek, o esnadaki kök adresi olan `450` geri döndürüyor.



Şekil 5.17 İkili arama ağacında ikinci durum görünümü.

Şekil 5.17'de görülen ikinci durum için fonksiyonu tekrar çalıştıralım. `5` silinecek ve bu düğümün adresi `200`, kökün adresi de `450`'dir. İlk olarak kökün adresiyle fonksiyon çağrılıyor;


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

Kökte `7` var, aranan düğüm henüz bulunamadığı 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);
```

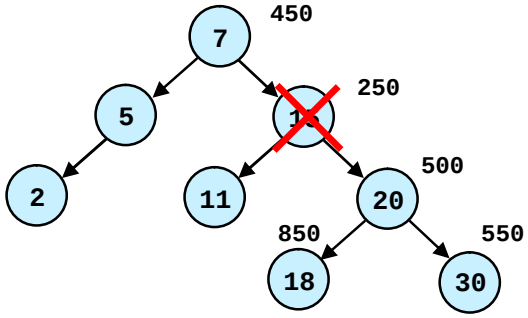
Aranan düğüm şimdi bulunuyor. Bu düğümün sol çocuğu var fakat sağ çocuğu yoktur. Bu yüzden aşağıda gösterilen `else` bloğu çalışıyor.

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

çağrısıyla 2 numara ile gösterilen bölümde çalışmıştı ve `return` komutu icra edilerek `p` geri döndürülmüştü. Öyleyse ilk çağrıldığı nokta olan 1.bölüme 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 Şekil 5.18'de görüldüğü gibi üçüncü durum için fonksiyonu adım adım çalıştırıyoruz.



Şekil 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 çağrılıyor;

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

1

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

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

2

Aranan düğüm bulundu. Yani `if(root->data == x)` koşulundaki `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 düğümün sağ soyundaki en küçük `data`'yı barındıran düğümün adresi `p` işaretçisine atanıyor. Bu işlem `while` döngüsü içerisinde `15`'in sağ çocuğu olan `20`'nin sol tarafındaki yaprağa kadar inilerek sağlanıyor. Şimdi `p` işaretçisinde bu en küçük veriyi barındıran düğümün adresi, `q` işaretçisinde ise silinecek düğümün 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 düğümün sol çocuğu olarak bağlanıyor. Ardından `15` siliniyor ve `q` işaretçisi geri döndürülüyor. Fonksiyon en son `delete_node(250, 15)` çağrısıyla 2 numara ile gösterilen bölümde çalışmıştı ve `return` komutu icra edilerek `q` geri döndürülmüştü. Şimdi ilk çağrıldığı nokta olan 1.bölüme 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 Searh Tree*) ikili arama ağacıdır.

İkili Arama Ağacında Bir Düğümü 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;
}
```

İkili Arama Ağacı 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;
}
```

İkili Arama Ağacında Minimum Elemanı Bulmak

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

İkili Arama Ağacında Maximum Elemanı Bulmak

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

Verilen İki Ağacı Karşılaştırmak

```
int isSame(BTREE* a, BTREE* b) {
    if(a == NULL && b == NULL)
        return 1; // İki ağaç 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 doğru ise true döndürür
        );
    else
        return 0;
}
```

Alıştırmalar

Örnek 5.5: Girilen bir `x` değeri, kökten itibaren yaprak dahil olmak üzere o yol üzerindeki verilerin toplamına eşitse `true`, eşit değilse `false` döndüren `path` isimli programı kodlayınız.

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

    if(root == NULL) // Ağaç 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)
        );
    }
}
```

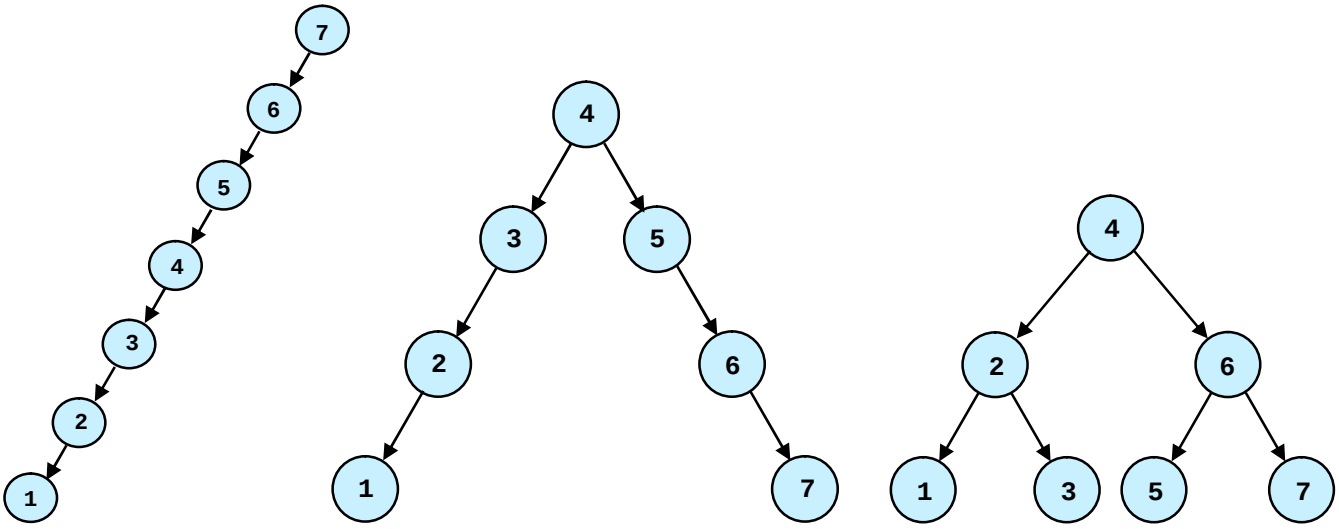
Örnek 5.6: Bir ikili arama ağacındaki verilerden tek olanları diğer bir BST ağacına kopyalayan copyOdd isimli programı yazınız.

```
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 hesaplanmış karmaşıklık hesapları ve yürütme zamanı bağıntılarından sapılır. Dolayısıyla ağaç veri modelinin en önemli getirisi kaybolmaya başlar. Aşağıda 1-7 arası sayıların farklı sırada ikili arama ağacına eklenmesiyle oluşan üç değişik ağaç gösterilmiştir.



Şekil 5.19 a) İkili bir arama ağacı. **Şekil 5.19 b)** İkili başka bir arama ağacı. **Şekil 5.19 c)** Daha başka bir ikili arama ağacı.

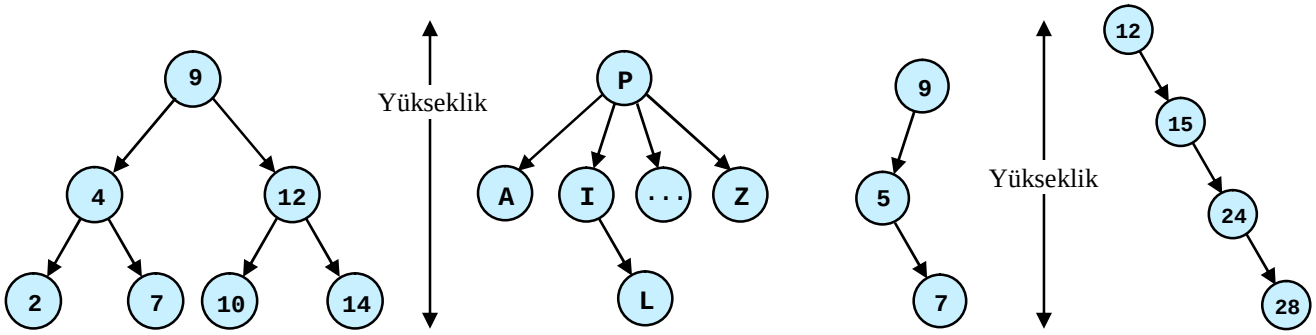
Sayıların hangi sırada geleceği bilinemediğinden uygulamada bu üç ağacın biri ile karşılaşılabilir. Bu durumda dengeli ağaçlar tercih edilir. Dolayısıyla ağacın dengesi bozulduğunda ağacı yeniden dengelemek gerekir. İkili arama ağacının yüksekliğinin olabildiğince düşük olması arzu edilir. Bu yüzden 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 $\theta(lgn)$ kalmasını sağlar.

$$Balance\ Factor = Heig\square t_{(right\ subtree)} - Heig\square t_{(left\ subtree)}$$

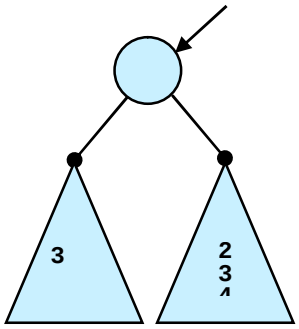
Denge faktörü -1, 0 ve 1 değerini alabilir. Dikkat edilmesi gereken önemli bir nokta herhangi bir düğümün denge faktörü hesaplanırken sol ve sağ ağaçların yüksekliklerinin belirlenmesidir. Yoksa herhangi bir kayıt için düğümlerin sayılması değildir. Bir başka deyişle AVL ağacında sol alt ağaç ile sağ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

$$h_{left} - h_{right} \leq 1$$



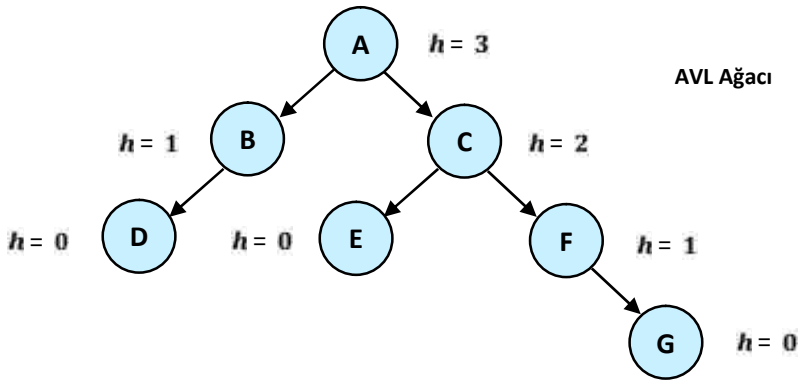
Şekil 5.20 a) Dengeli bir ikili ağaç. **Şekil 5.20 b)** Dengeli bir ağaç. **Şekil 5.20 c)** Dengesiz ikili ağaçlar.

Sağ veya sol çocuk yok ise (*yani olmayan düğümlerin ya da diğer bir söyleyişle `NULL` değerlerin*) yüksekliği `-1`'dir. Şekil 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.



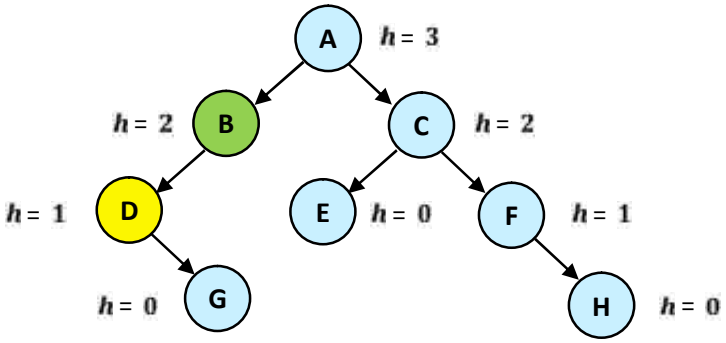
Şekil 5.21 `AVL` ağacında sol ve sağ alt ağaçların yüksekliği.

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



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

Şimdi de `AVL` olmayan bir ağacı inceleyelim. Şekil 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 aradaki fark `2`'dir. Bu nedenle bir `AVL` ağacı değildir. Şekildeki yeşil renkli düğüm gibi tek çocuğu olan ve aynı zamanda torunu da olan ağaçlar bir `AVL` ağacı değildir de denilebilir.



Şekil 5.23 Dengeli olmayan ikili bir ağaçta düğümlerin yükseklikleri.

Önerme:

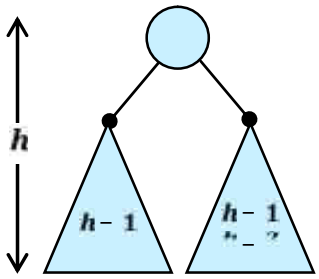
Bir ikili ağacın yüksekliği $lgn < h < n$ iken, n düğümlü bir AVL ağacının yüksekliği $\theta(lgn)$'dir. Burada θ, gayri resmi olarak mertebe ya da civarı anlamındadır (θ *gösteriminin formal tanımı, Algoritmalar dersinde verilecektir*). Öyleyse ifadeyi, bir AVL ağacının yüksekliği 2 tabanında logaritma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

$n = 10^6$ ise $lg10^6 \approx 20$ civarındadır.

Bu önerme AVL ağaçları için bir kolaylık sağlamaktadır. Yüksekliğin az olması dengeyi sağlamakta 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 $h \geq 2$ için, h yüksekliğindeki bir AVL ağacı, kök ve $h-1$ yüksekliğinde bir alt ağaç ile yine $h-1$ veya $h-2$ yüksekliğinde diğer bir alt ağaç içerir.



Şekil 5.24 AVL ağacında alt ağaçların yükseklikleri.

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

İspat: Bir AVL ağacının yapısı Şekil 5.24'te ki gibiyse bu ağaçta bir kök, sol alt ağaç ve sağ alt ağaç olacağından;

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

yazabiliriz. Sağ alt ağaç $f(h-1)$ veya $f(h-2)$ yüksekliğinde olabilir. Fakat tanımdaki minimum olma gereğinden dolayı $f(h-2)$ alınmıştır. O halde $f(h-1) \geq f(h-2)$'dir. Eşitsizlik $f(h) \geq 1 + 2f(h-2)$ biçiminde de yazılabilir. Düzenlendiğinde;

$$f(h) > 2f(h-2)$$

olur. $f(h)$'taki h yerine eşitsizliğin sağında yer alan $h-2$ yazıldığında bu defa sağ taraf $2.2f(h-2-2)$'den $4f(h-4)$ olur. Yine h yerine $h-2$ koyarsak $2.4f(h-2-4)$'ten $8f(h-6)$ olur. Bu işlemleri tekrarladığımızda ise baştan itibaren yeniden yazarsak;

$$f(n) > 2f(n-2)$$
$$f(n) > 4f(n-4)$$
$$f(n) > 8f(n-6)$$
$$f(n) > 16f(n-8)$$
...
...

şeklinde devam edecektir. Dikkat edilirse, eşitsizliğin sağındaki fonksiyonun katsayıları 2'nin kuvvetleri şeklinde devam etmektedir. Fakat kaça kadar devam edeceği belli olmadığından katsayıyı 2^i şeklinde ifade edebiliriz. Ayrıca h'tan çıkartılan sayıların 2'nin kuvveti olan i sayısıyla yine 2'nin çarpımları şeklinde devam etmektedir. İfadeyi i iterasyon kadar devam ettirirsek;

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

Daha önce f(0)'ın 1'e eşit olduğunu belirtmiştik. Şimdi eşitsizliğin sağ tarafındaki h − 2i, 0'a eşitlendiğinde, h − 2i = 0 eşitliğinden i = h/2 bulunur. Bulunan $n/2$'yi üstteki ifadede yer alan i'nin yerine yazıldığında ise;

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

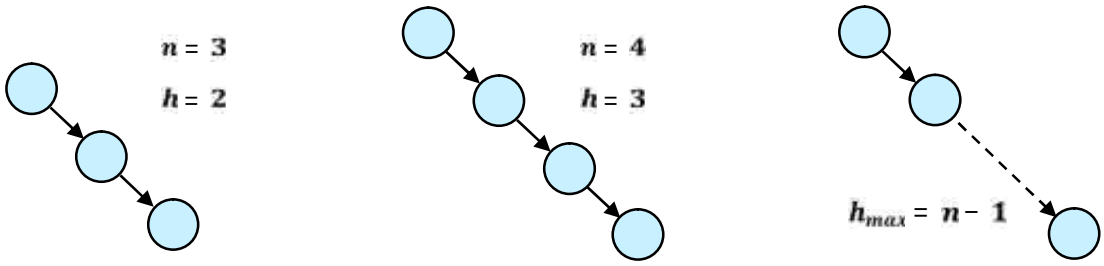
elde edilir. Her iki tarafın 2 tabanında logaritması alınırsa;

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

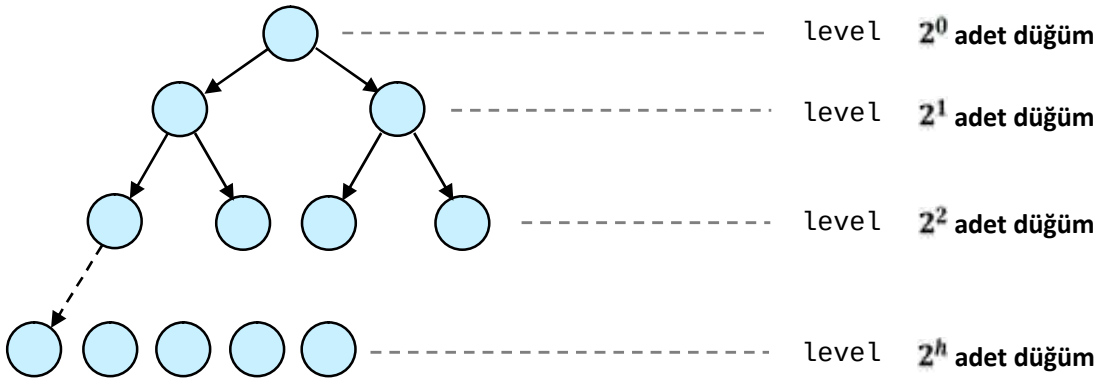
olur. Buradan da $h < 2\lg f(n)$ yazılabilir. n düğümlü bir ikili ağacın yüksekliği en az $\lg n$ olacağından, AVL ağacının yüksekliği $\theta(\lg n)$'dir.

Örnek 5.7: n düğümlü bir ikili ağacın maksimum ve minimum yüksekliğini bulunuz.

Çözüm: İkili ağacın maksimum yüksekliğini bulmak oldukça kolaydır. Bir ikili ağacın maksimum yüksekliğe sahip olabilmesi için aşağıdaki şekilde görüldüğü gibi düğümlerin ardı ardına sıralanması gerekir.



Şekilde 3 düğümlü bir ikili ağacın yüksekliği 2, 4 düğümlü bir ikili ağacın yüksekliği ise 3'tür. O halde n düğümlü bir ağacın maksimum yüksekliği $h_{max} = n - 1$'dir. Bir ikili ağaçta minimum yüksekliği sağlamak için bir seviye tamamlanmadan diğer seviyeye düğüm bağlanmamalıdır. Mümkün olduğu kadar ağacı sıkışık yapmak gerekmektedir. Böyle bir ikili ağaçta bir düğümün kök düğümden olan uzaklığına düzey (*level*) dendiği bilindiğine göre kök düğümün düzeyi 0 (*sıfır*), yaprakların düzeyi ise h'a eşit olacaktır.



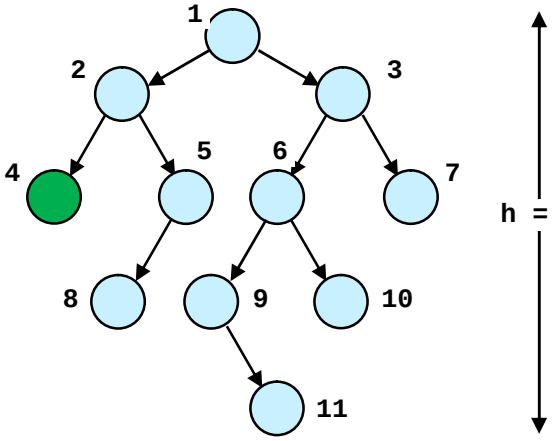
Son düzeydeki düğüm sayısının en fazla 2^h adet olabileceğine dikkat ediniz. Öyleyse bu şekilde bir h yüksekliğindeki ikili ağaçta olabilecek maksimum düğüm sayısı, $1 + 2 + 2^2 + \cdots + 2^h$ geometrik serisi olacaktır. Bu da $2^{h+1} - 1$'e eşittir. İkili bir arama ağacındaki n adet düğümle h yükseklik ilişkisini $2^{h+1} - 1 \geq n$ şeklinde ifade edebiliriz. Düzenlendiğinde ise, $2^{h+1} \geq n + 1$ şeklini alan ifadenin her iki tarafının 2 tabanında logaritmasını aldığımızda;

$$h + 1 \geq \lg n + 1 \quad \Rightarrow \quad h \geq \lg n + 1 - 1$$

olur. Buradan ikili bir ağacın minimum yüksekliği $h_{min} = \lg n + 1 - 1$ 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 değişmektedir.

Bir AVL Ağacının Yapısı

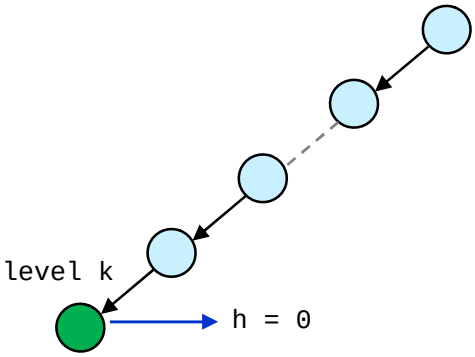
n düğümlü bir AVL (*dengeli bir* BST) ağacı düşünelim. Köke en yakın yaprak k seviyesinde olsun. Köke en yakın yaprak k seviyesindeyken bir ağacın yüksekliği $k+1$, $k+2$, $k+3$... değerlerini alabilirken en fazla $2k$ değerinde olabilir. Şekil 5.25'de görülen AVL ağacında köke en yakın yaprak 4 no'lu 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. Şimdi bu düğümün en yakın yaprak olduğunu varsayalım. En yakın yaprak demek, diğer yapraklar daha üst seviyede değil, daha üstte daha yakın yapraklar yok demektir. Peki, bu ağacın yüksekliği kaç olabilir? Tabii ki en az 2 olmalıdır. Evet ama en fazla ne kadar olabilir? Bir AVL ağacı çizerek konuyu daha iyi kavrayalım.



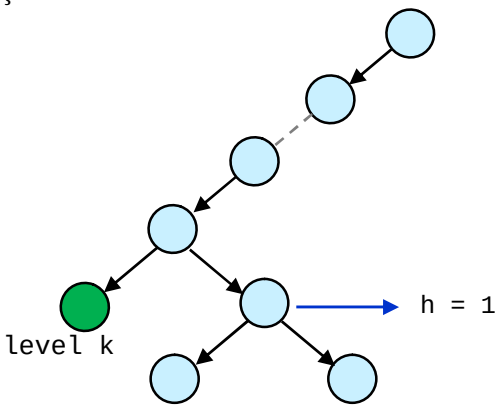
Şekil 5.25 AVL ağacının 2k yüksekliğinde olması.

Şekildeki AVL ağacında köke en yakın yaprak 4 numaralı yeşil renkli olan yapraktır. Yüksekliği 0'dır. Kardeş 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 olacağı 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 değeri değişecek, onlara da ekleme yapılırsa köke en yakın yaprak da değişmiş olacaktır. Bir AVL ağacının yüksekliğinin en fazla 2k olduğunun ispatı aşağıda görülüyor.

İspat: Yeşil renkli düğüm şu anda `level k` düzeyinde ve amaç, ağacın maksimum yüksekliğe ulaşmasını sağlamak olsun. Ağacın üst seviyelerindeki düğümler ve dallar gösterilmemiştir.



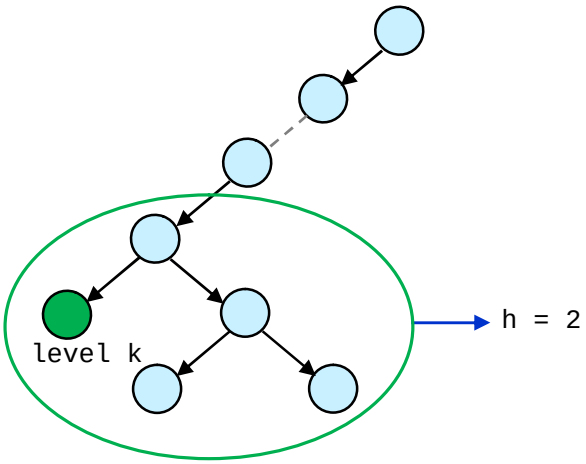
Şekil 5.26 a) AVL ağacında k seviyesindeki yaprak.



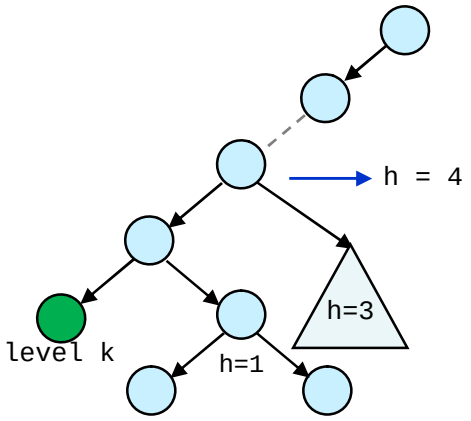
Şekil 5.26 b) k seviyesinde bir yaprak varken ekleme yapmak.

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 değişecektir. 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

ağaç 2 yüksekliğindeyse denge gereği sağ alt ağacın yüksekliği **en fazla** 3 olabilir. Sağ alt ağacın bağlı olduğu düğümün yüksekliği ise 4'tür.

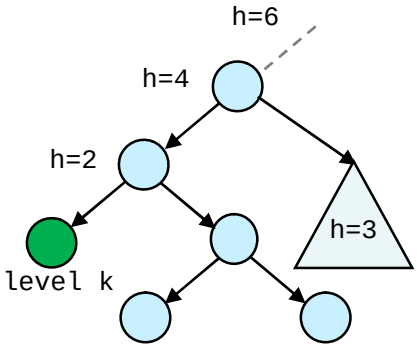


Şekil 5.27 a) Daire içine alınmış alt ağacın yüksekliği 2'dir.



Şekil 5.27 b) Üçgenle gösterilen alt ağacın yüksekliği 3'tür.

Bir üst seviyeye çıktığımızda ise sağ alt ağacın yüksekliği **en fazla** 5 olabilirken bağlı olduğu ebeveyninin yüksekliği 6'dır ve bu şekilde devam eder.



Şekil 5.28 Köke en yakın yaprak ve atalarının yükseklikleri.

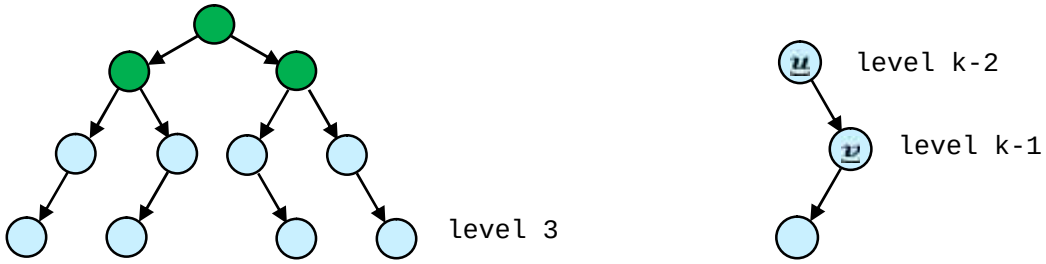
Kökün yüksekliği bilinmediği için x'le gösterilirse aşağıdaki tablo ortaya çıkacaktır. Şekil 5.28'deki örnekten devam edersek, köke en yakın yaprak k seviyesindeyken yüksekliği 0'dır. Yaprağın 2 yüksekliğindeki ebeveyninin seviyesi ise k-1'dir. Dikkat edilirse yaprağın her atasının yükseklikleri 2'şer artmaktayken, seviye ise 1 azalmaktadır. Toplamda 0.seviye ile birlikte k+1 kadar sayı vardır. Öyleyse yükseklikteki x'in değeri 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

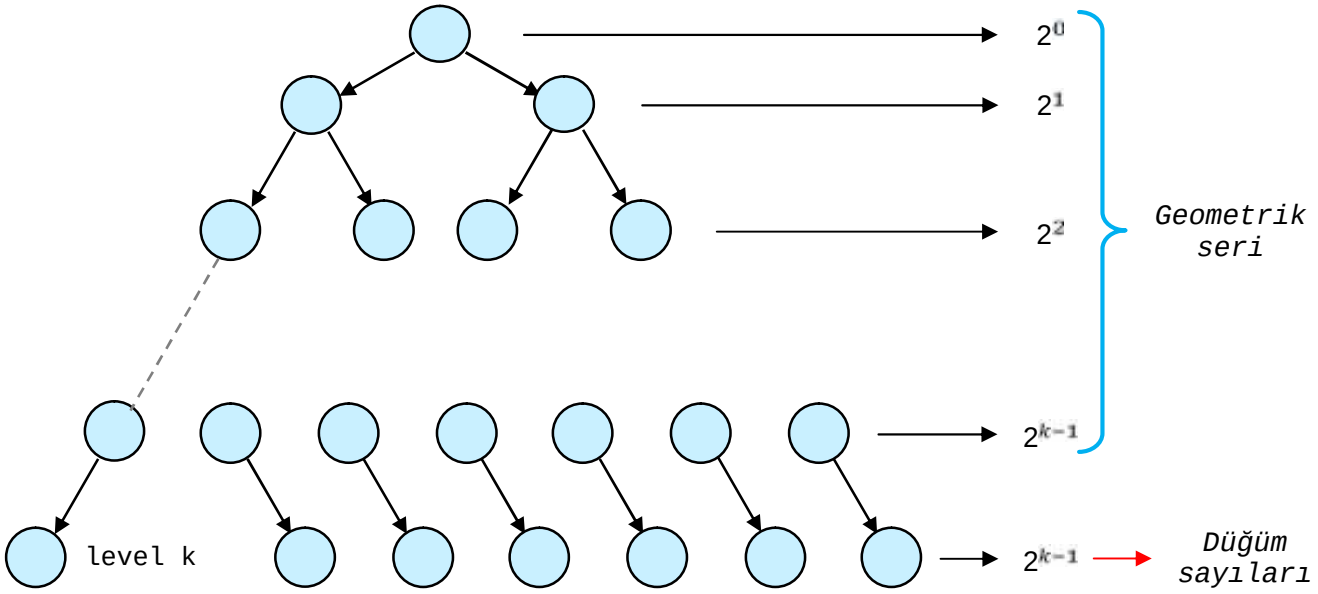
İddia: Köke en yakın yaprak k seviyesinde ise 0, 1, 2, ..., k-2 seviyelerindeki düğümlerin 2 çocuğu vardır.

İspat: Bu iddiayı ispatlamak için çelişki yöntemini kullanabiliriz. k-2 seviyesindeki u düğümünün sadece bir v çocuğu olduğu kabul edilsin. O halde v, k-1 seviyesindedir ve yaprak olamaz. Bu yüzden v düğümünün **en az** bir çocuğu vardır. Bir çocuğunun olması ise u düğümünde bir yükseklik ihlali demektir ve çelişki elde edilir.



Şekil 5.29 k-2 seviyelerindeki düğümlerin 2 çocuğu vardır.

Bu iddia `0, 1, 2, ..., k-1` seviyelerinin tamamen dolu olduğunu göstermektedir. Köke en yakın yaprak `k` seviyesindeyken ağaçtaki **minimum** eleman sayısı da hesaplanabilir. Aşağıdaki şekilde hesaplama mantığı gösterilmiştir.



Şekil 5.30 Köke en yakın yaprak `k` seviyesindeyken ağaçtaki minimum düğüm sayısı.

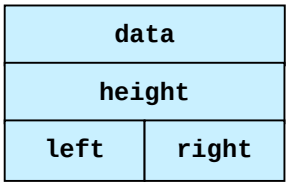
Şekil 5.30'da görüldüğü gibi ağaçtaki düğüm sayısının **minimum** olabilmesi için, köke en yakın yaprak düzeyinden daha aşağısında düğüm olmaması gerekir. `k` düzeyinde ise yine **minimum** olma özelliğinden dolayı birer yaprak bulunmalıdır ve bu da ağacın `k` ve `k-1` düzeylerinde aynı sayıda düğüm olması demektir. 2^0, 2^1, 2^2, ... , 2^{k-1} toplamları geometrik bir seridir ve bu toplam $2^k - 1$ sayısına eşittir. Ağaçtaki minimum düğüm sayısı ise bu geometrik seri ile `k` düzeyinin düğüm sayısının toplamına eşittir.

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

Artık bir AVL ağacının genel yapısını tanımlayabiliriz.

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

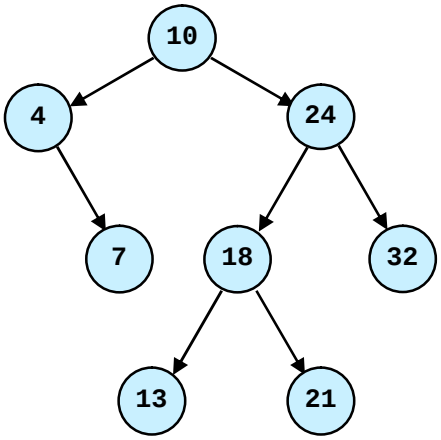
`typedef` bildirimi yaparak `struct node` yerine bundan sonra AVLTREE kullanacağız.



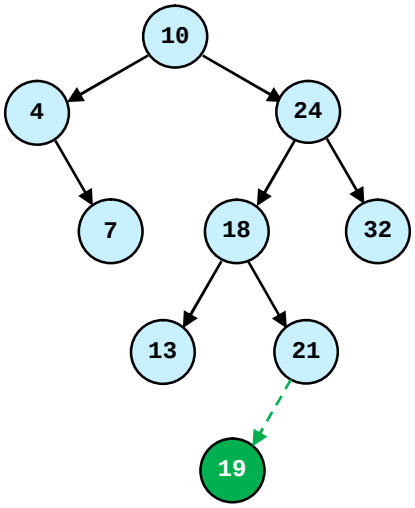
Şekil 5.31 Bir AVL düğümünün mantıksal gösterimi.

AVL Ağaçlarında Ekleme İşlemi

AVL ağaçlarına ekleme yapılırken hem denge hem de BST özelliği kaybolmamalıdır. Ekleme işleminde bazı düğümlerin yükseklikleri değişebileceğinden ağacı tekrar denge durumuna ayarlamak gerekir. Aşağıda Şekil 5.32 a)'da BST tarzında bir AVL ağacı görülüyor. Aynı ağaca Şekil 5.32 b)'de görüldüğü üzere 19 içeriğinin eklenmek istenmesi halinde ağaçta denge kavramı ortadan kalkıyor. 24 içeriğinin bulunduğu düğümün yüksekliği ile 4 içeriğinin bulunduğu düğümün yüksekliği arasındaki fark 2'ye çıkıyor. Daha önce anlatmış olduğumuz ikili arama ağaçlarına düğüm ekleme işlemindeki gibi rekürsif bir fonksiyonla AVL ağacını denge durumuna sokmaya çalışmak oldukça maliyetlidir. Örneğin bir ağaçta 1 milyon adet veri bulunabilir. Bu kadar veriyi düzenlemeye çalışmak çok zahmetli olacaktır.

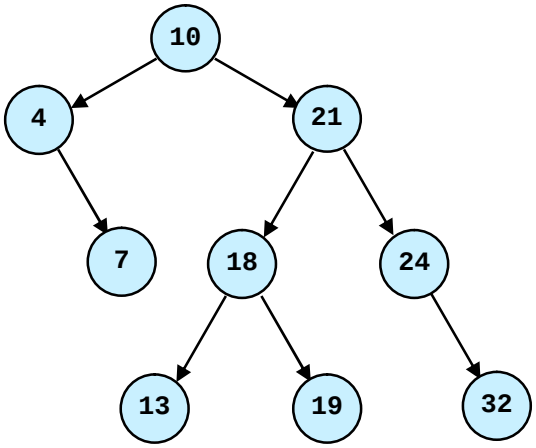


Şekil 5.32 a) Henüz dengede olan bir AVL ağacı.



Şekil 5.32 b) 19 içeriğinin eklenmesiyle denge bozuluyor.

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



Şekil 5.33 19 eklendikten sonra döndürme uygulanmış AVL ağacı.

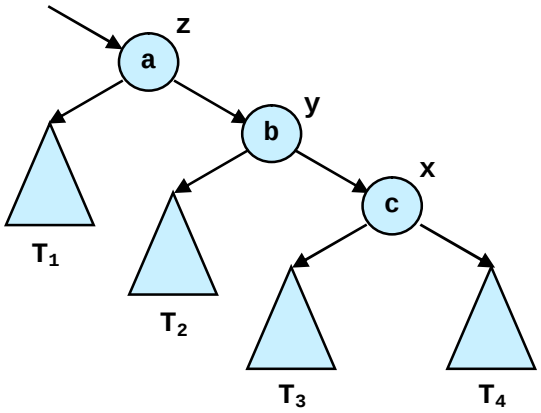
AVL ağaçlarında ekleme işleminde ağacı döndürmek gerektiğinden ekleme fonksiyonunu düğümleri döndürme konusunu anlattıktan sonra yazacağız.

Bir AVL Ağacında Düğümleri Döndürmek

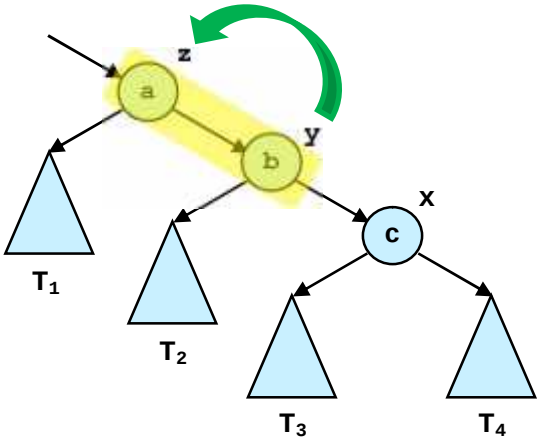
AVL ağaçlarında düğümleri döndürmek için ana olarak iki tane döndürme işlemi olmakla birlikte 4 farklı yol bulunmaktadır. İkisi **single** (*tek*) rotation ve diğer ikisi de **double** (*çift*) rotation işlemidir. Her iki döndürme biçimi aslında aynı olmakla birlikte birbirinin simetriğidir ve biri sol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

Tek Döndürme (*Single Rotation*)

Şekil 5.34 a)'da görüldüğü gibi alt ağaçları olan bir AVL ağacında $\mathbf{T_3}$ 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** düğümüne **x** ve arada kalan düğüme de **y** adını vereceğiz. Hemen akla **a**'nın birden çok torunu olduğu ve neden **c** düğümüne **x** adının verildiği akla gelebilir. Nedeni basittir, çünkü düğüm ekleme işlemi $\mathbf{T_3}$ tarafına yapılmıştır ve ihlalin olduğu yol üzerinde bulunan torun **c** düğümüdür. **x**, **y** ve **z** olarak isimlendirilen düğümler **zig-zig** durumundadır. Böyle bir durumda **z** düğümüne sola döndürme (*left rotate*) işlemi uygulanır. Sola döndürmenin nasıl yapılacağı Şekil 5.34 b)'de görülmektedir. **b** düğümünden itibaren saat yönünün tersine yani sola doğru 90° döndürülür.



Şekil 5.34 a) Denge durumunda bir AVL alt ağacı.

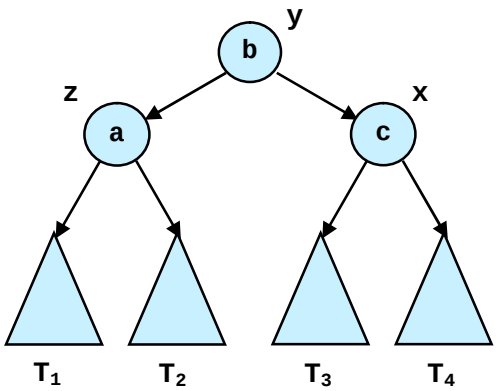


Şekil 5.34 b) Ekleme yapıldığında zig-zig yapısında ki AVL alt ağacı.

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 Şekil 5.35'te görülmektedir. Döndürme işleminden sonra diğer alt ağaçların yerleştirilmesi BST'ye uygun biçimde yapılmalıdır. Yukarıdaki şekle göre alt ağaçlar;

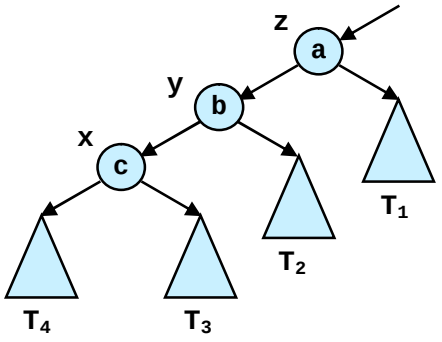
$$T_1 \leq z \leq T_2 \leq y \leq T_3 \leq x \leq T_4$$

şeklinde sıralanırlar. Burada alt ağaçlar sıralı bir biçimde dizilmiştir fakat her zaman bu şekilde sıralı olmayabilir.

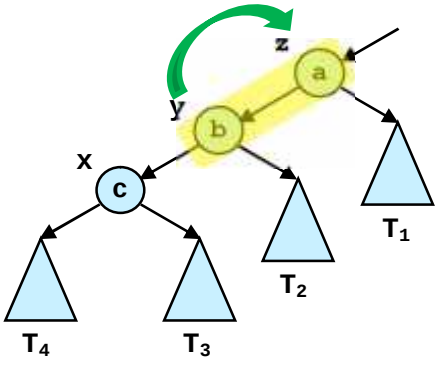


Şekil 5.35 Sola döndürülmüş AVL alt ağacı.

Şimdi de tek döndürmenin (*single rotation*) simetriği olan sağa döndürme (*right rotate*) işlemini inceleyelim. Şekil 5.36 a)'da yer alan T_3 ya da T_4 alt ağaçlarından birine ekleme yapılmak istensin. İhlalin görüldüğü düğümü yine z olarak isimlendireceğiz ve ihlalin olduğu düğümden ekleme yapılan düğüm üzerindeki yol üzerinde bulunan toruna da x ismini vereceğiz. Arada kalan çocuk düğüm ise yine y adını alacaktır.



Şekil 5.36 a) Denge durumunda bir AVL alt ağacı.

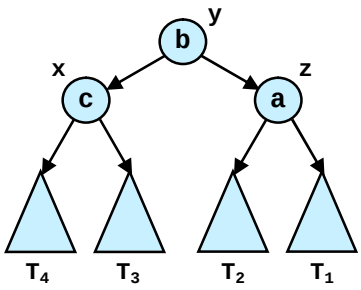


Şekil 5.36 b) Döndürme uygulanacak zig-zig yapısında AVL alt ağacı.

Bu durumda da z düğümüne sağa döndürme (*right rotate*) işlemi uygulanır. Sağa döndürmenin nasıl yapılacağı Şekil 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;

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

şeklindedir. Alt ağaçların son durumu Şekil 5.37'de görülüyor.

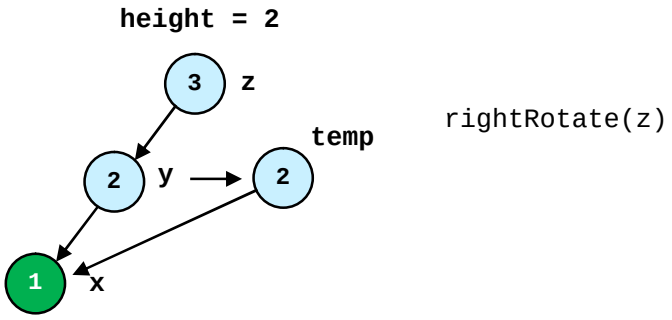


Şekil 5.37 Sağa döndürülmüş AVL alt ağacı.

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

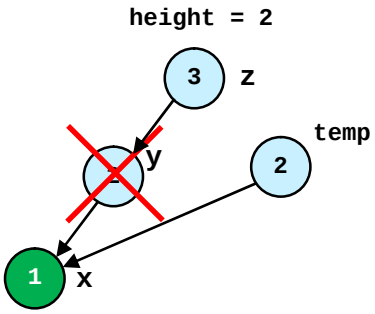
```
AVLTREE *rightRotate(AVLTREE *z) {
    AVLTREE* temp = z -> left; // z düğümünün sol çocuğu temp'e atanıyor
    z -> left = temp -> right; // temp'in sağ çocuğu z'nin sol çocuğu olarak atanıyor
    temp -> right = z; // son olarak z düğümü temp'in sağ çocuğu olarak atanıyor
    // Yükseklikler update ediliyor
    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 döndürülecek düğümün adresini almaktadır. Şekil 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* türden bir değişken oluşturuluyor ve y adını verdiğimiz z düğümünün sol çocuğu bu değişkene atanıyor. Şu anda hem y düğümünün, hem de `temp`'in sol çocuğu yeni eklenen 1 düğümüdür.

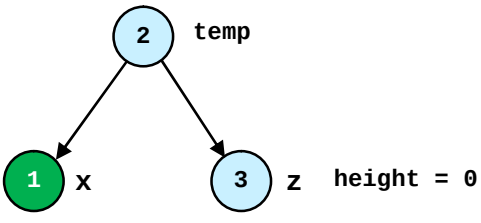


Şekil 5.38 z düğümünün ilk baştaki yüksekliği 2'dir.

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



Şekil 5.39 a) z ile y düğümü arasındaki bağ koparılıyor.



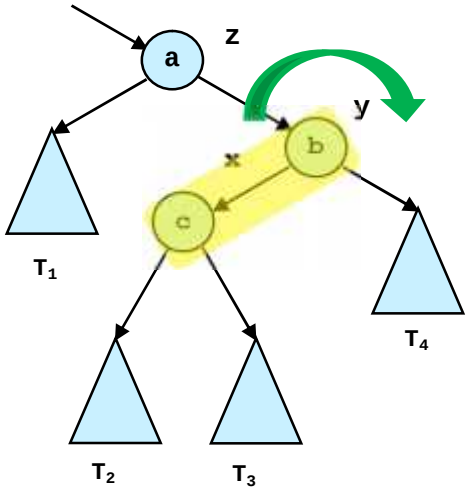
Şekil 5.39 b) x ve z artık `temp`'in çocukları durumundadır.

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

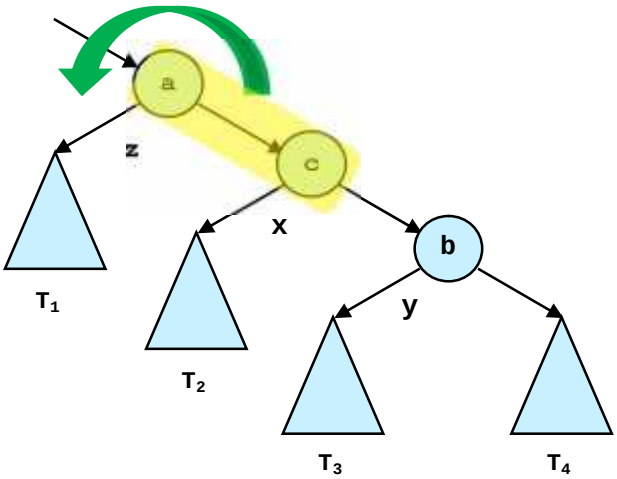
```
AVLTREE *leftRotate(AVLTREE *z) {
    AVLTREE *temp = z -> right; // z düğümünün sağ çocuğu temp'e atanıyor

    z -> right = temp -> left; // temp'in sol çocuğu z'nin sağ çocuğu olarak atanıyor
    temp -> left = z; // son olarak z düğümü temp'in sol çocuğu 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 işlemindeki gibi çift döndürme işleminde de ekleme yapıldığı zaman oluşan ihlalin görüldüğü ilk düğüme z adı verilir. İhlalin görüldüğü düğümden ekleme yapılan düğüme doğru olan yol üzerinde bulunan toruna da bildiğiniz gibi x ismi ve arada kalan çocuk düğüme de y ismi verilmektedir. Şekil 5.40 a)'da **zig-zag** tipinde bir alt ağaç görülmektedir. Bu alt ağaçta T_2 ya da T_3'den birine ekleme yapıldığında AVL özelliği ihlalinin görüldüğü a düğümü z olarak, diğer düğümler de y ve x olarak adlandırılmıştır. Double rotate mantığını daha iyi anlamak için iki kural iyi bilinmelidir. İlki, ağaç b düğümünden itibaren BST özelliği de dikkate alınarak bir kez sağa döndürülmeli ve **zig-zig** durumuna getirilmelidir. İkinci olarak ise ağaç sağa döndürüldükten sonra z olarak adlandırılan a düğümünden itibaren sola döndürülmelidir.

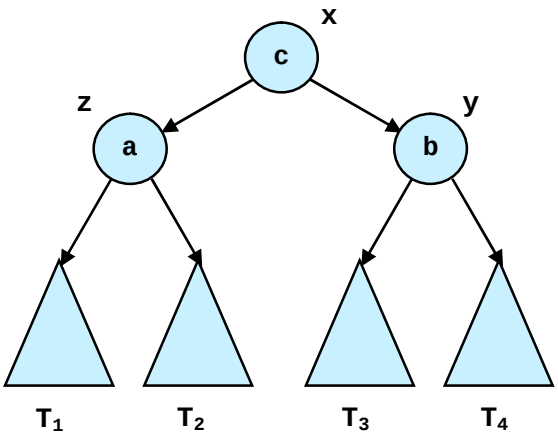


Şekil 5.40 a) Ağaç sağa döndürülmelidir.



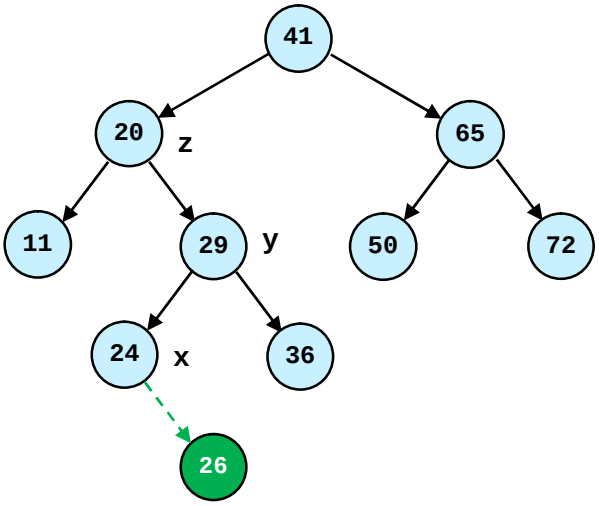
Şekil 5.40 b) Ağaç şimdi sola döndürülmelidir.

Şekil 5.40 a)'da ağacın ilk durumu, Şekil 5.40 b)'de ise bir kez sağa döndürülerek **-right rotate R-R(b)-** zig-zig biçimine getirilmiş ağacı görüyorsunuz. Ağaç bu kez sola döndürülerek **-left rotate L-R(a)-** double rotation işlemi tamamlanmış olacaktır. Ağacın en son durumu Şekil 5.41'de görülüyor.



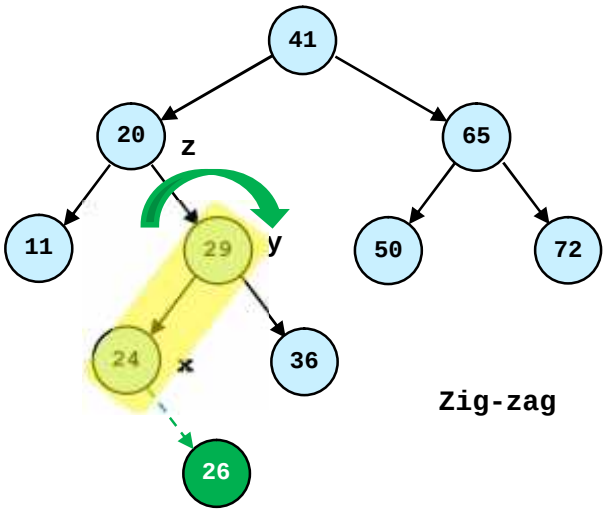
Şekil 5.41 Double rotation uygulanmış alt ağaç.

Double rotation'ın simetriği olan döndürme yönteminde de farklı bir işlem yoktur. Bir örnekle çift döndürme işleminin simetriğini de inceleyelim. Örnek ağacımız Şekil 5.42'deki gibi olsun.

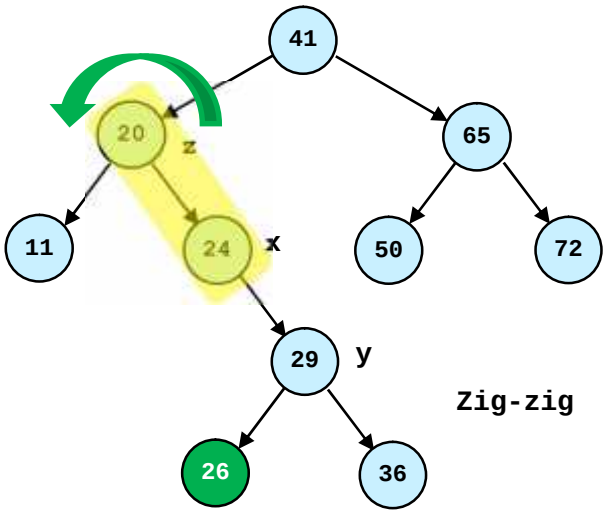


Şekil 5.42 Alt ağaca 26 eklenmek isteniyor.

Şekilde görülen ağaca 26 sayısı eklenmek istendiğinde ilk olarak BST'ye uygun bir biçimde ağacın neresine ekleneceğini belirlemek gerekmektedir. Belirleme yapıldıktan sonra eklenmiş olan düğümden itibaren köke olan yol üzerinde ilerleyerek denge ihlalinin ilk görüldüğü düğüm olan 20'ye z adı veriliyor. z'nin o yol üzerindeki torununa x ve arada kalan düğüm de y olarak isimlendiriliyor. Meydana gelen yol **zig-zag** biçiminde olduğundan y olarak adlandırılan düğümün Şekil 5.43 a)'da olduğu gibi sağa doğru döndürülmesi gerekmektedir. Şekil 5.43 b)'de ise **zig-zig** durumunda ki ağaç sola döndürülerek denge sağlanmalıdır.

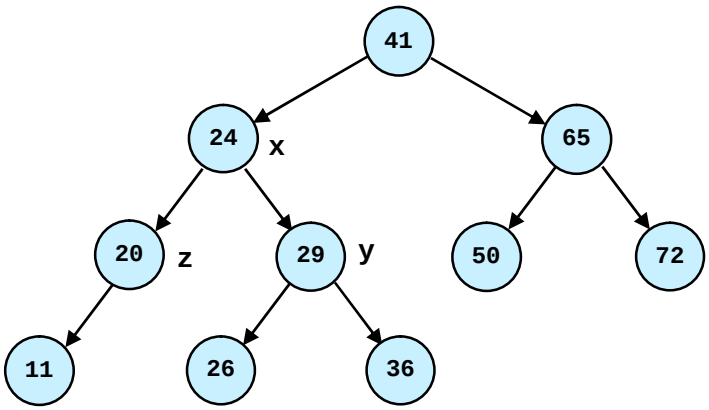


Şekil 5.43 a) Ağaç ilk olarak sağa döndürülmelidir.



Şekil 5.43 b) Ağaç son olarak sola döndürülmelidir.

Şekil 5.44'te double rotation yöntemiyle ekleme yapılarak BST ve AVL özelliği korunmuş olan ağaç görülmektedir.



Şekil 5.44 Double rotation uygulanmış AVL ağacı.

Double rotation için `leftRightRotate` ve `rightLeftRotate` fonksiyonları aşağıdaki gibi tanımlanmıştır. Fonksiyonların her ikisi de ihlalin görüldüğü `z` düğümünü parametre olarak alıyor. Ağaç `y` düğümünden itibaren sola, ardından da sağa döndürülüyor.

```
AVLTREE *leftRightRotate(AVLTREE *z) {
    z -> left = leftRotate(z -> left); // y düğümü sola döndürülüyor
       /* Fonksiyondan geri dönen x düğümü, z düğümünün sol çocuğu olarak
         atanıyor. Aynı zamanda y düğümünün ebeveyni durumunda */

    return rightRotate(z); // z düğümü bu defa sağa döndürülüyor
}
```

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

```
AVLTREE *rightLeftRotate(AVLTREE *z) {
    z -> right = rightRotate(z -> right); // y düğümü sağa döndürülüyor
       /* Fonksiyondan geri dönen x düğümü, z düğümünün sağ çocuğu olarak
          atanıyor. Aynı zamanda y düğümünün ebeveyni durumunda */

    return leftRotate(z); // z düğümü bu defa sola döndürülüyor
}
```

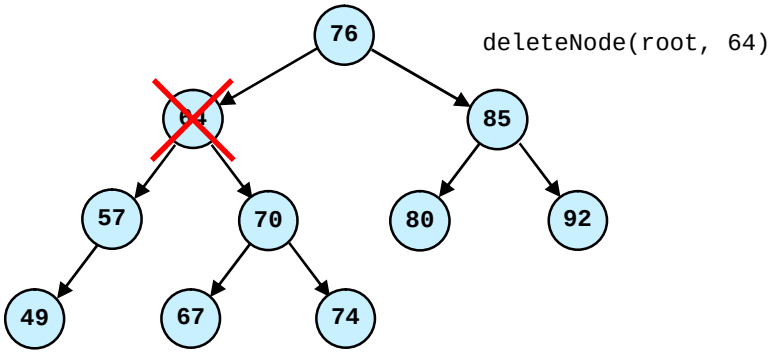
Döndürme kodlarını yazdığımıza göre bir `AVL` ağacına ekleme işleminin fonksiyonunu da yazabiliriz. Fonksiyon parametre olarak eklenecek veriyi ve ağacın adresini almaktadır. Geri dönüş değeri ise gerekli döndürmeler uygulanarak `AVL` özelliği sağlanmış yeni ağacın köküdür.

```
AVLTREE *insertToAVL(int x, AVLTREE *tree)
{
    if(tree != NULL)
    {
        if(x < tree -> data) // eklenecek verinin kökten büyüklüğüne 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 düğümün adresi geri döndürülüyor

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

        /* Eğer balans faktör 1'den büyükse ve eklenen veri ağacın sol çocuğunun
           datasından küçük ise ağaç sağa döndürülmelidir. */
        if((height(tree->left) - height(tree->right)) > 1 && x < tree->left->data)
            return RightRotate(tree);

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

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

        /* Eğer balans faktör -1'den küçükse ve eklenen veri ağacın sağ çocuğunun
           datasından küçük ise ağaç ilk olarak sağa 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 Ağaçlarında Silme İşlemi

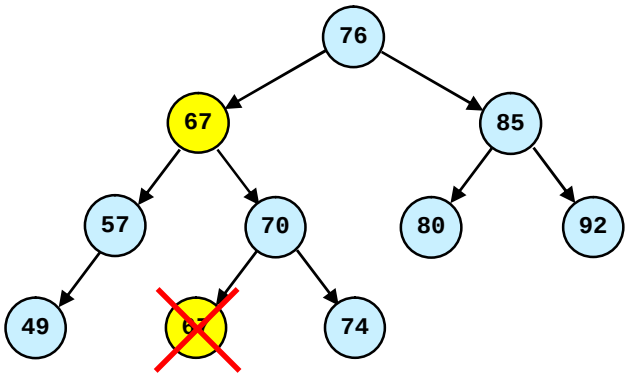
AVL ağaçlarında bir düğümü silme işlemi yapılırken hem denge özelliği, hem de BST özelliği kaybolmamalıdır. Silme işleminde eklemede olduğu gibi bazı düğümlerin yükseklikleri değişebileceğinden ağacı tekrar denge durumuna getirmek gerekir. Bir düğüm eklendikten hemen sonra silindiğinde oluşan AVL ağacı, bir öncekinin aynısı değildir. Bir düğümü silmek istediğimizde eğer ağaçta hiç eleman yoksa fonksiyon NULL ile geri dönmelidir. Ağaçta eleman var ise, ancak aranılan değere bir eşitlik varsa silme işlemi gerçekleşecektir. Silinecek düğüm ağacın çocuklarından herhangi biri ise ve verisi, üzerinde bulunduğumuz 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 bulunduğunda silme işleminde de 3 durumdan bahsedilebilir. Bunlar; eğer silinecek düğümün hiç çocuğu yoksa bu düğüm yapraklardan biridir ya da köktür. Düğüm direk 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. Şayet silinecek düğümün iki çocuğu var ise sağ alt ağacındaki en küçük veriyi barındıran düğüm bulunarak silinecek düğüme kopyalanır ve kopyalanan sağ alt ağaçtaki en küçük veriyi barındıran düğüm silinir. Denge bozulmuşsa gerekli döndürmeler yapılarak silme işlemi tamamlanmış olur. Bir örnekle konuyu daha iyi kavrayalım.

Aşağıda görülen Şekil 5.45'teki AVL ağacından 64'ün silinmek istendiğini kabul edelim.



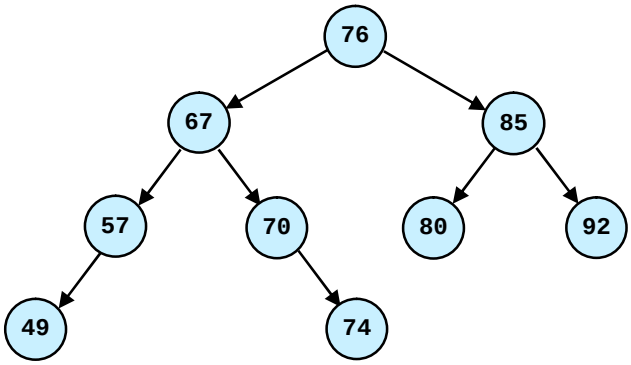
Şekil 5.45 AVL ağacından 64 değeri silinmek isteniyor.

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



Şekil 5.46 Şimdi alt ağaçtaki 67 değeri silinmelidir.

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



Şekil 5.47 Silme işleminden sonra ağacın son şekli.

Ağaçta denge bozulmadığı için herhangi bir döndürme uygulanmamıştır. Eğer silinen alt ağaçtaki kopya düğümün de çocukları olsaydı, yukarıda konu girişinde anlattığımız kurallar tekrar uygulanacaktı. Şimdi bir AVL ağacından silme işlemini gerçekleştiren `deleteNode` isimli fonksiyonu yazacağız fakat ilk olarak, `deleteNode` fonksiyonu içerisinde kullanacağımız sol alt ağaç ile sağ alt ağaç yüksekliklerinin farkını hesaplayan `getBalance` fonksiyonunu yazmamız gerekiyor.

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

Şimdi bir AVL ağacından silme işlemini gerçekleştiren `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 değerini 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 ağacında en küçük veriye sahip olan düğüm bulunuyor
            root -> data = temp -> data;
            // bulunan minimum değer silinecek düğüme kopyalanıyor
            root -> right = deleteNode(root -> right, temp -> data);
            /* Artık silinecek değer temp'tedir. Fonksiyon kendisini sağ çocuğu ve
               Silinecek temp değeriyle tekrar çağırıyor */
        }
    }
    if (root == NULL)
        return root;
    // Ağacın yüksekliği güncelleniyor
    root -> height = max(height(root -> left), height(root -> right)) + 1;
    // Eğer balans 1'den büyükse ve sol alt ağacın 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 ağacın 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 ağacın 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 ağacın 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 ÖNCELİKLİ KUYRUKLAR (*Priority Queues*)

Öncelikli kuyruklar (*priority queues*), terim anlamıyla gündelik yaşamda sık karşılaştığımız bir olguyu belirler. Bazı durumlarda bir işi öteki işlerin hepsinden önce yapmak zorunda kalabiliriz. Örneğin, bir fatura ödeme veznesinde kuyruğa girenler arasında öncelik sırası önde olanındır. Ancak, bir kavşakta geçiş önceliği cankurtaranındır. Bir hava meydanına iniş sırası bekleyen uçaklar arasında, öncelik sırası acil iniş isteyen uçağındır. Bir hastanede muayene önceliği ise durumu en acil olan hastanındır.

Bilgisayarlarda öncelikli kuyrukların kullanımına örnek olarak printer ve işletim sistemi verilebilir. Ağ yazıcılarında'da az sayfaya sahip belge önce yazılır. Çok sayfası olan bir belge yazdırılmak üzere yazıcıya gönderilmiş ve hemen akabinde iki sayfalık bir belge daha yazdırılmak istenmiş olabilir. İki sayfalık belgenin çıktısını almak için sayfalar dolusu belgenin yazdırma işleminin bitmesi beklenmez. Arada bir bölümde iki sayfa yazdırılır. İşletim sisteminde ise en kısa zamanlı işlem önce çalışır. Gerçek zamanlı işlemler de öncelik değerine sahiptir.

Görüldüğü gibi, bir koleksiyon içinde öncelik sırasını farklı amaçlarla belirleyebiliriz. En basiti, ilk gelen ilk çıkar dediğimiz FIFO (*first in first out*) yapısıdır. Ama bu yapı karşılaşılacak bütün olasılıklara çözüm getiremez. Dolayısıyla, koleksiyonun öğelerini istenen önceliğe göre sıralayan bir yapıya gereksinim vardır. Biraz genellemeyle, öncelikli kuyruklar yapısına da kuyruk diyeceğiz; ama burada yüklenen anlamı FIFO yapısından farklı olabilir. Son giren de ilk çıkabilir, ilk giren de ilk çıkabilir. Duruma göre değişmektedir. Önceliği en yüksek olan kuyruktan ilk çıkar. Soyut bir veri tipi (ADT – *Abstract Data Type*) olarak öncelikli kuyruklarda sadece sayılar ya da karakterler değil, karmaşık 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 kuyruğa eklenme zamanı gibi elemanların alanları ile ilgili olmayan dışsal bir değere göre de sıralı olabilirler.

Öncelikli kuyruklar, temel kuyruk işlemlerinin sonuçlarını elemanların gerçek sırasının belirlediği veri yapılarıdır. Azalan ve artan sırada olmak üzere iki tür öncelik kuyruğu vardır. Artan öncelik kuyruklarında (*min heap*) elemanlar herhangi bir yere eklenebilir ama sadece en küçük eleman çıkarılabilir. Azalan öncelik kuyruğu (*max heap*) ise artan öncelik kuyruğunun tam tersidir. Artan öncelik kuyruğunda önce en küçük eleman, sonra ikinci küçük eleman sırayla çıkarılacağından dolayı elemanlar kuyruktan artan sırayla çıkmaktadırlar. Birkaç eleman çıkarıldıktan sonra ise daha küçük bir eleman eklenirse doğal olarak kuyruktan çıkarıldığında önceki elemanlardan daha küçük bir eleman çıkmış olacaktır.

Kuyruktaki her nesnenin karşılaştırılabilir bir öncelik değeri (*key*) vardır. Bazı sistemlerde küçük değer önceliklidir, bazı sistemlerde de küçük değerler, küçük öncelikli olabilir. İlk nesne kuyruğun başına konulur ve önceliği belirten bir key değeri atanır. Öncelikli kuyruklarda da diğer kuyruklarda olduğu gibi ekleme ve silme işlemleri mevcuttur.

Bu kuyrukları bilgisayarda nasıl implemente edebiliriz? Daha önceki kuyrukları bağlı listelerle ve dizilerle implemente etmiştik. Peki, burada ne yapmalıyız? Bağlı listelerde işlem oldukça kolaydı ve listenin sonuna ekleyip, baştan çıkarma işlemi yapılıyordu. Burada ise öncelik durumu varsa başa eklenmeli, küçük öncelikli ise araya eklenmelidir. Numarasına göre farklı yerlere atanmalı ya da önceliği yüksek olan kuyruktan çıkmalıdır.

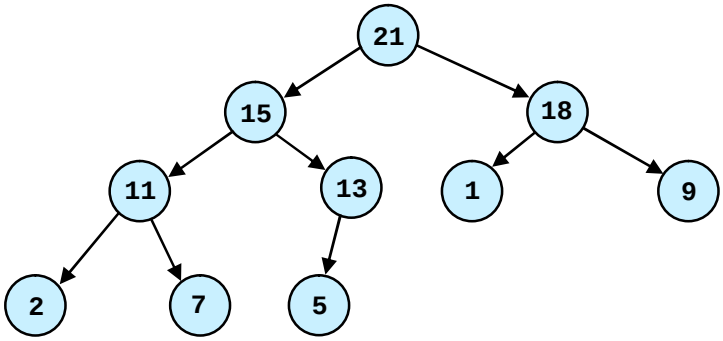
Tüm bu kuralların uygulandığı bir dizi kullandığımızı varsayalım ve dizinin içindeki değerlerin de öncelikli değerler (*bu öncelikli değer de küçük değer olsun*) olduğunu kabul edelim. Bu durumda en yüksek öncelikli değeri kuyruktan çıkarmak için dizi üzerinde arama yapmak gerekmektedir. Bir dizi üzerindeki en küçük değeri 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 (*İkili Yığın*)

Yığın (*heap*) yapısı n tane elemandan oluşan ve öncelikli kuyrukları gerçekleştirmek için kullanılan ikili ağaç şeklinde bir yapıdır. İkili arama ağacı, arama işlemini hızlı yapmak için üretilmiş bir veri yapısı iken yığın, bir grup eleman arasından önceliği en yüksek olan elemanı hızlı bulmak ve bu elemanı yapıyı bozmadan hızlıca silmek için üretilmiş bir veri yapısıdır. İkili arama ağacında her düğüm sol alt ağacındaki tüm düğümlerden büyük, sağ alt ağacındaki tüm düğümlerden küçük ya da eşit iken, yığında her düğüm sol ve sağ alt ağacındaki tüm düğümlerden büyüktür ya da küçüktür. Sol ve sağ alt ağaç arasında büyüklük küçüklük ilişkisi açısından bir zorlama yoktur. Ayrıca ikili arama ağacında düğümlerin sol veya sağ çocukları olabileceği gibi hiç çocuğu olmayabilir. Yığında ise elemanlar seviye seviye yerleştirilir. Üstteki seviye dolmadan alttaki seviyede eleman bulunmaz. Dolayısıyla son seviyeden önceki seviyedeki tüm düğümlerin iki çocuğu vardır. Son satır dolu olmayabilir fakat soldan sağa doğru doldurulmalıdır.

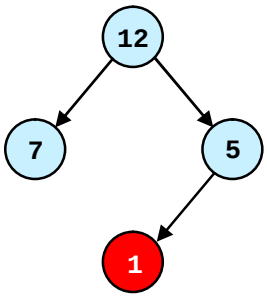
Yığınlarda en büyük değerli eleman ağacın kökündeyse bu dizilime **max-heap**, en küçük değerli eleman ağacın kökündeyse bu dizilime **min-heap** denir. Max heap'te her bir düğümün değeri, çocuklarının değerlerinden küçük olamaz. Büyüktür ya da eşittir. Min heap'te ise her bir düğümün değeri, çocuklarının değerlerinden küçüktür veya eşittir, büyük olamaz. Örneğin beş sayfalık bir belgeyi yazıcıya gönderdiğimizde bir öncelik değeri atanacaktır. Ardından beş sayfalık başka bir belge daha yazıcıya gönderildiğinde bu belgeler de ilk belgelerin öncelik değerine sahip olacaktır, yani öncelik değerleri eşittir.

Şekil 5.48'de görülen max-heap ağacı ikili bir ağaçtır (**BT**-*binary tree*) fakat ikili arama ağacı (**BST**-*binary search tree*) değildir. Her düğüm kendi çocuklarından büyüktür.



Şekil 5.48 Max-Heap ağacı.

Aşağıdaki ağaç ise bir heap değildir. Çünkü her seviye dolu olmasına karşın, son seviyenin doldurulmasına en soldan başlanmamıştır.



Şekil 5.49 Heap özelliği olmayan bir ağaç.

Heap, bir ağaçla temsil edilebilir. Ağaç da doğrusal olmayan bağlı bir listedir. Heap, liste şeklinde yapılabilir fakat listelerle yönetmek yerine dizilerle işlem yapmak daha kolaydır. Çünkü her seviye doludur ve son satır da soldan sağa doğru bir dizilime sahiptir. Bu nedenle yığınlar, genelde tek boyutlu dizilerle implemente edilirler.

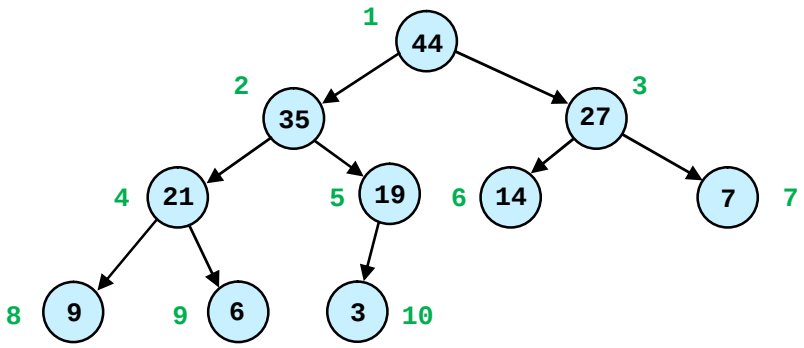
Mapping (*Eşleme*): Ağacın öncelikle kökü dizideki ilk elemandır. Heap, bir dizide tam ikili ağaç yapısı kullanılarak gerçekleştirildiğinde, dizinin `1.` elemanından başlar, `0.` indis kullanılmaz. Dizi sıralı değildir, yalnız `1.` indisteki elemanın en öncelikli (*ilk alınacak*) eleman olduğu garanti edilir. Dizinin bir `i` indisindeki elemanın ebeveyninin (*parent*) indisi `i/2`'ye eşittir.

```
parent(i) =  i/2
```

Buradaki köşeli paranteze benzeyen simge, içindeki ondalıklı sayısal değeri aşağı yuvarlamaktadır. Örneğin 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 sağ çocuktur. Tersten düşünürsek kökün indisi `1` ise, sol çocuğunun indisi `2i`, sağ çocuğun indisi ise `2i + 1`'dir.

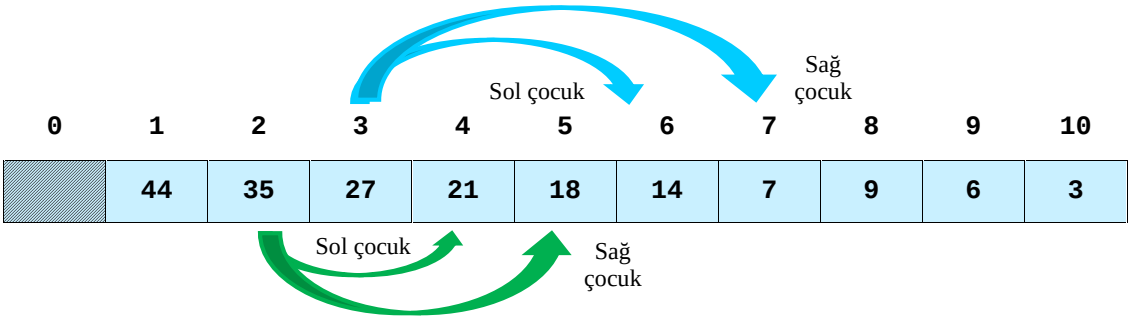
```
left(i)  = 2i
right(i) = 2i + 1
```

Şimdi heap'in dizi implementasyonunu Şekil 5.50 üzerinde inceleyelim. Ağaçtaki elemanların hepsini bir dizide saklayacağız. Ağaçta `10` eleman olduğuna göre dizi boyutu `11` olmalıdır. Çünkü `0` indisli dizinin ilk elemanının kullanılmadığını söylemiştik. Dolayısıyla veri ataması yapılmayacaktır.



Şekil 5.50 Örnek heap ağacı.

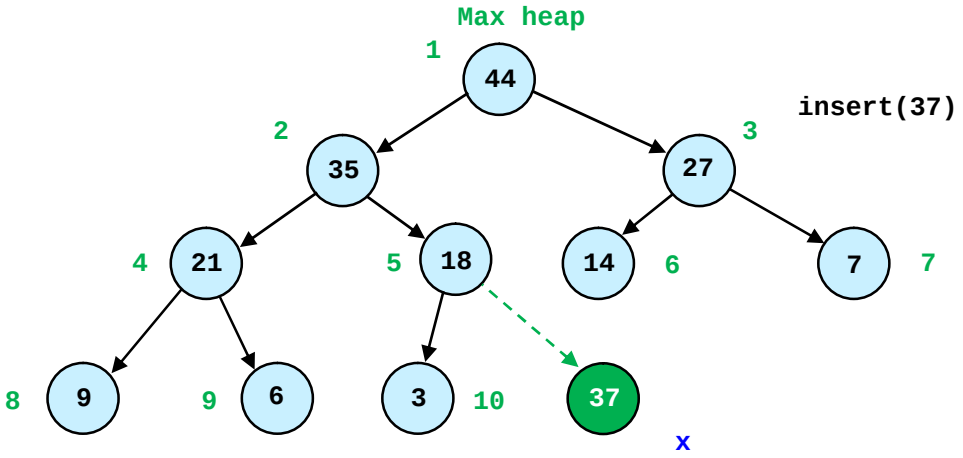
Heap'in dizi ile implemente edilmiş hali Şekil 5.51'de görülmektedir. Dizide 2. indiste bulunan elemanın sol çocuğu 4. indisteki 21, sağ çocuğu ise 5. indisteki 18'dir. Yine aynı şekilde 3 numaralı indisteki elemanın sol çocuğu 6. indiste yer alan 14 ve sağ çocuğu da 7. indisteki 7'dir.



Şekil 5.51 Heap'in dizi uygulaması.

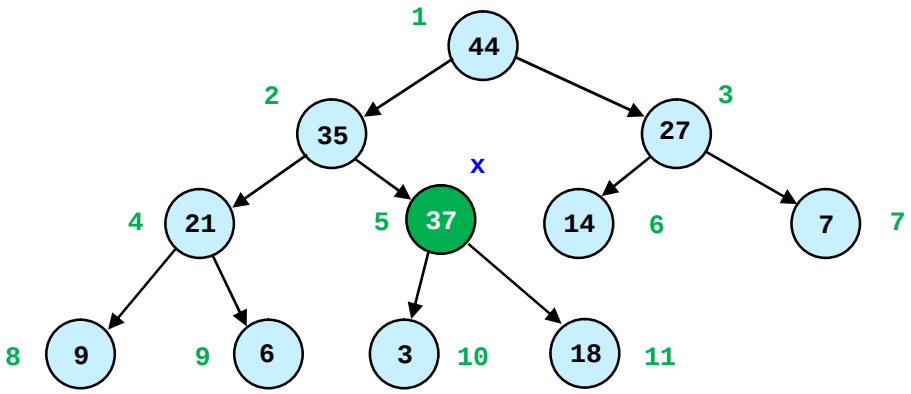
Heap İşlemleri

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



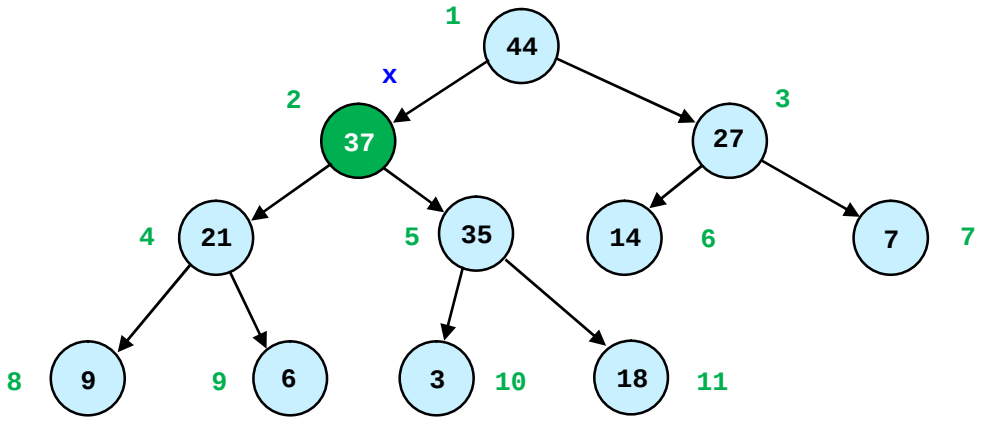
Şekil 5.52 a) Heap'e 37 sayısı ekleniyor.

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



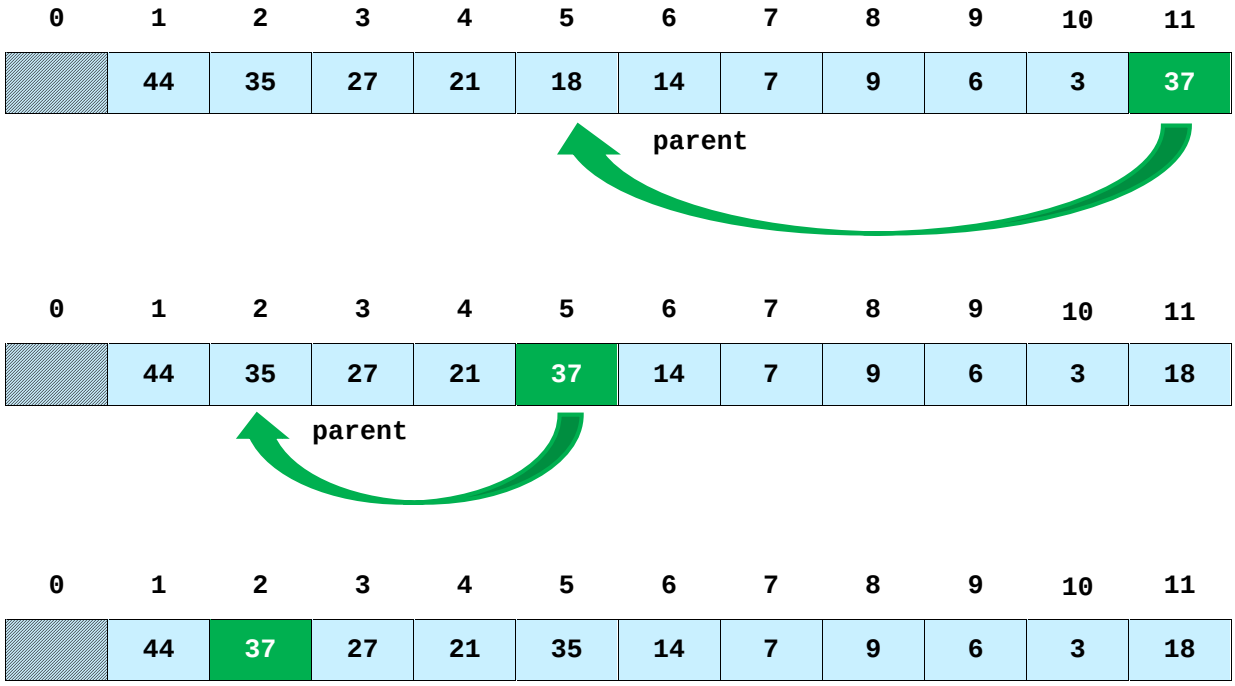
Şekil 5.52 b) 37 sayısı ebeveyniyle yer değiştiriyor.

x artık ebeveyni olan 44'ten büyük olmadığı için işlem sona erer. Ekleme işleminden sonra heap ağacının son durumu Şekil 5.52 c)'de görülüyor.



Şekil 5.52 c) 37 sayısı bir kez daha ebeveyniyle yer değiştiriyor.

Burada yapılan işlemlere **yukarıya doğru tırmanma** (*percolate up*) denir. Ekleme işleminin dizi uygulaması Şekil 5.53'te gösterilmiştir.



Şekil 5.53 37 sayısı heap'e ekleniyor. Ardından ebeveyniyle karşılaştırılıyor ve eğer ebeveyninden büyükse yer değiştiriyor.

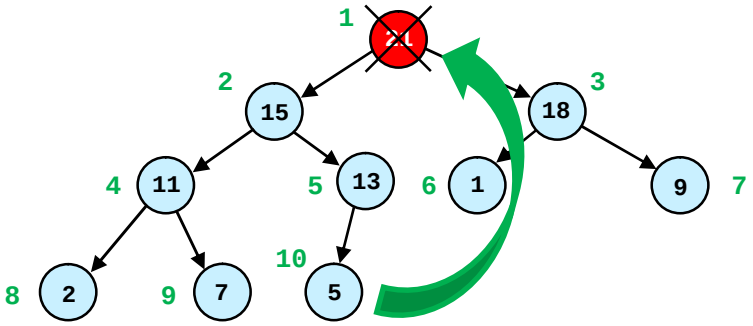
Tanımlanmış bir max heap dizisine değer ekleyen `insertToMax_heap` fonksiyonu aşağıdaki gibi yazılabilir. Fonksiyon parametre olarak eklenecek heap'in adresini, eklenecek değeri ve heap'in sıradaki boş indeksini almaktadır. Geriye bir değer döndürmeyeceği için türü `void` olarak tanımlanmıştır. Fonksiyon içerisinde çağrılan `swap` fonksiyonunu artık biliyorsunuz. `swap()`, kendisine gelen iki değişkenin değerlerini birbirleriyle değiştirmektedir.

```
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 değiştiriliyor
            index /= 2;
        }
    }
}
```

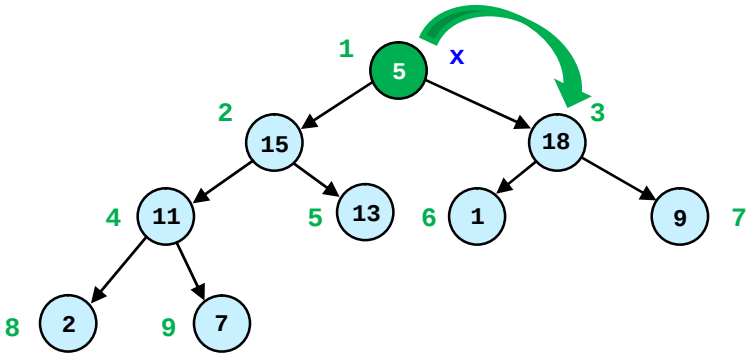
Aynı fonksiyon ufak bir oynama ile min heap için de yazılabilir. `while` koşulu içerisindeki küçüktür `(<)` işaretini büyüktür `(>)` işaretine çevirmek ve fonksiyonun adını değiştirmek yeterlidir.

2- Delete (*Silme*): Bir max heap'ten maksimum eleman silindiğinde 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 değer silinir. Yerine ağacın en alt satırının en sağındaki `x` olarak adlandırılan düğüm konulur. Eğer `x` çocuklarının değerlerinden küçük ise en büyüğü ile `swap` yapılır. Bu işlem x çocuklarından küçük olmayıncaya kadar devam eder.

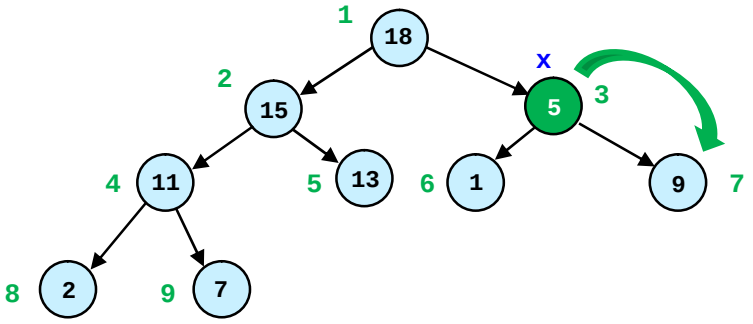
Şekil 5.54'te silme işleminin basamakları gösterilmiştir. Silme fonksiyonu `21` değerini geri döndürür ve heap'ten siler. Yerine son satırının en sağındaki veriyi atar ve çocuklarıyla teker teker karşılaştırılır. Eğer küçükse en büyüğüyle yeri değiştirilir ve bu işlem küçük olmayıncaya kadar devam eder.



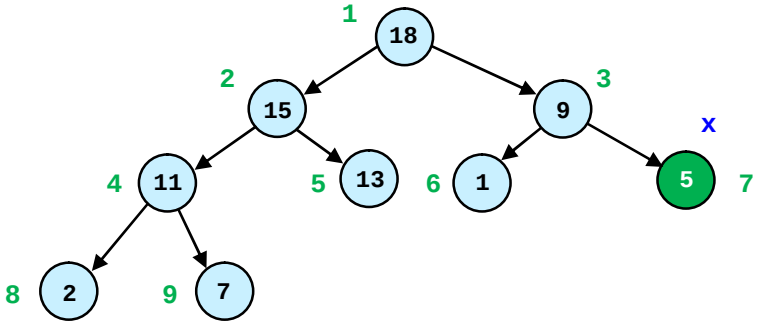
Şekil 5.54 a) `21` sayısı heap'ten siliniyor.



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



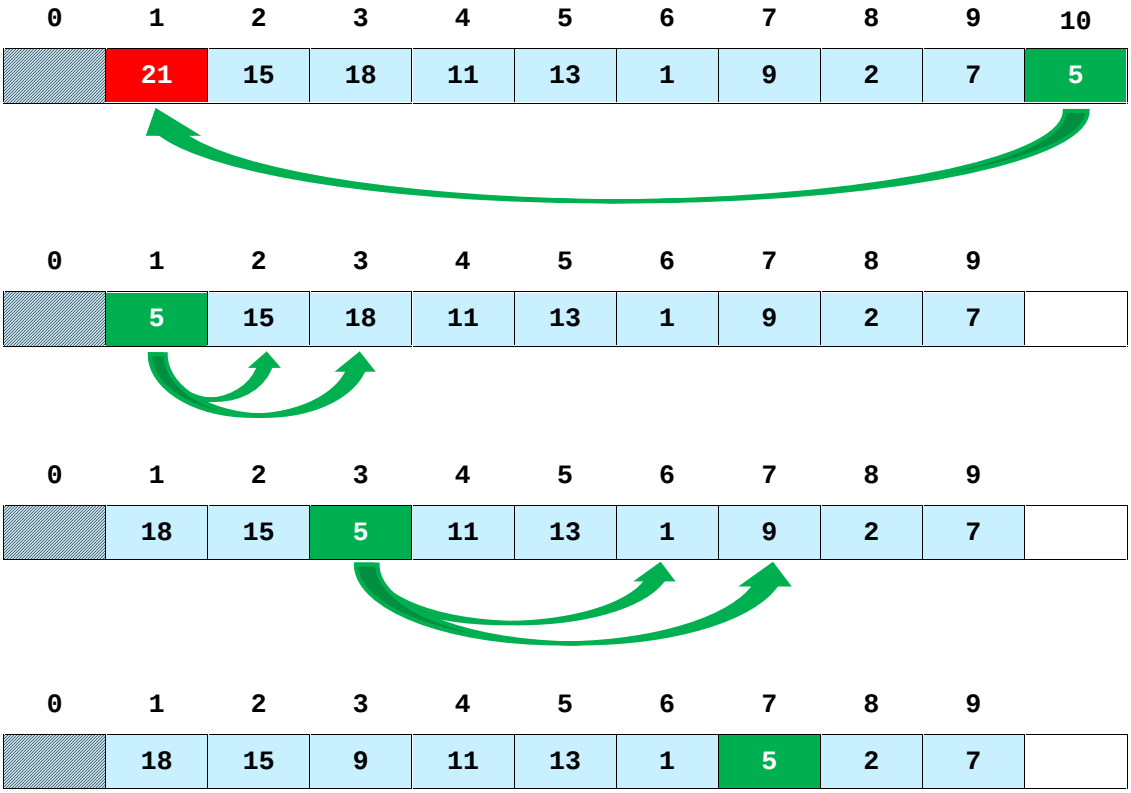
Şekil 5.54 c) `5` değeri çocuklarıyla karşılaştırılıyor ve büyük olanla yer değiştiriyor.



Şekil 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 maksimuma bir kerede ulaşma imkânı verir. Burada ağacı tekrar

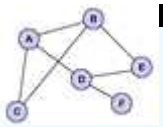
düzenlemenin elbette bir maliyeti vardır. Kökten başlayarak en alttaki yaprağa kadar bir dolaşma gerektirir ve bu dolaşma da ağacın yüksekliği ile orantılıdır. Peki n düğüme sahip bir heap'in yüksekliği nedir? Yükseklik lgn civarındadır. Örneğin `1` milyon tane verisi olan heap'te silme işleminden sonra en fazla `20` iterasyon yapmak yeterlidir. Silme işleminde max heap'deki yukarı tırmanma işleminin tersi bir durum vardır. Bu işleme **perculate down** denmektedir. Yapılan `delete` işleminin dizi uygulaması Şekil 5.55'te görülmektedir.



Şekil 5.55 Silme işleminin dizi uygulaması.

Şimdi max heap'ten bir elemanı silen `deleteMax` ismli fonksiyonu yazalım. Fonksiyon parametre olarak, elemanı silinecek heap'in adresini ve heap'in ekleme yapılabilecek boş indeksini almaktadır. Silinen elemanın değeriyle 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 değiştiriliyor
                i = 2*i;
            }
            else { // sağ çocuk büyükse
                swap(&array[(2*i) + 1], &array[i]); // sağ çocukla yer değiştiriliyor
                i = (2*i) + 1;
            }
        }
        return max;
    }
}
```



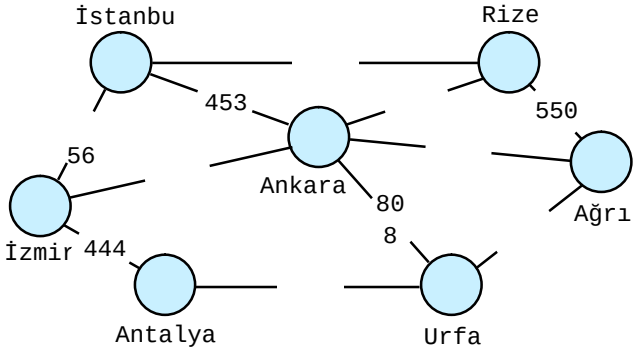
BÖLÜM

Graphs (*Çizgeler*) 6

6.1 GİRİŞ

Çizge kuramının başlangıcı Königsberg'in 7 köprüsü (*Kaliningrad/Rusya*) problemine dayanır. 1736'da Leonhard Euler'ın söz konusu probleme ilişkin kullandığı sembol ve çizimler graf kuramına ilişkin başlangıç olarak kabul edilir. Problem, "Pregel nehri üzerinde iki ada birbirine bir köprü ile, adalardan biri her iki kıyıya ikişer köprü, diğeri de her iki kıyıya birer köprü ile bağlıdır. Her hangi bir kara parçasında başlayan ve her köprüden bir kez geçerek başlangıç noktasına dönülen bir yürüyüş (*walking*) var mıdır?" sorusunu içeriyordu. Euler kara parçalarını düğüm, köprüleri de kenar kabul eden bir çizge elde etti. Söz konusu yürüyüşün ancak her düğümün derecesinin çift olması durumunda yapılabileceğini gösterdi. Graf kuramının bu ve buna benzer problemlerle başladığı düşünülür. Günlük yaşamdaki birçok problemi, graf kuramı uzayına taşıyarak çözebiliyoruz.

Graf (*çizge*), bilgisayar dünyasında ve gerçek hayatta çeşitli sebeplerle karşılaşılan bir olay veya ifadenin düğüm ve çizgiler kullanılarak gösterilmesi şeklidir. Fizik, kimya gibi temel bilimlerde, mühendislik uygulamalarında ve tıp biliminde pek çok problemin çözümü ve modellenmesi graflara dayandırılarak yapılmaktadır. Örneğin elektronik devreleri (*baskı devre kartları*), entegre devreleri, ulaşım ağlarını, otoyol ağını, havayolu ağını, bilgisayar ağlarını, lokal alan ağlarını, interneti, veri tabanlarını, bir haritayı veya bir karar ağacını graflar kullanarak temsil etmek mümkündür. Ayrıca planlama projelerinde, sosyal alanlarda, kimyasal bileşiklerin moleküler yapılarının araştırılmasında ve diğer pek çok alanda kullanılmaktadır. Örneğin Türkiye'de bazı illerin karayolu bağlantıları ile yolun uzunluğunu gösteren aşağıdaki yapı, yönsüz bir graftır.



Şekil 6.1 Yönsüz graf örneği.

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

Şekil 6.1'de verilen yapıyı grafın teknik deyimleri ile belirttiğimizde iki il arasındaki yollar **ayrıt** (*edge*), iki ayrıtın birleştiği yer **tepe** ya da **düğüm** (*vertex, node*) olarak anılır. Biz bu derste herhangi bir tepeyi (*düğüm*) küçük harf ve o tepenin indisi ile örneğin v_4 biçiminde göstereceğiz. Grafı oluşturan tepelerin sırası önemli olmaksızın yalnız bir kez yazıldığı kümeyi de (*set*) büyük harf V ile belirteceğiz. Örneğin yukarıdaki grafta V'nin elemanı olan tepeler (*vertices*);

v_1 : *İstanbul*
v_2 : *İzmir*
v_3 : *Ankara*
v_4 : *Antalya*
v_5 : *Urfa*
v_6 : *Rize*
v_7 : *Ağrı*

olarak belirlenirse,

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

Tek bir ayrıtı küçük harf e_i ile, ayrıtların oluşturduğu kümeyi (*set*) ise büyük harf E ile belirteceğiz. Her tepe bir bilgi parçasını gösterir ve her ayrıt iki bilgi arasındaki ilişkiyi gösterir. Eğer ayrıtlar üzerinde bir yön ilişkisi belirtilmemişse bu tür graflar **yönsüz graf** (*undirected graph*) adını alırlar. Yönsüz grafın ayrıtları bir parantez çifti içine o ayrıtın birleştiği tepeler yazılarak belirtilir. Örneğin (v_3, v_6) ayrıtı, ANKARA ile RİZE arasındaki yolu belirtmektedir. Yönsüz grafta 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 kuruluşunu sağlamak ve veri yapısını kurarken gözden kaçan bir ayrıt olmadığını daha kolay denetlemek için indisi daha küçük olan tepe önce yazılır. Bu yazım biçimine göre yukarıdaki örnekte ayrıt 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, band genişliğini belirten bir ağırlıktır.

Aynen ağaçlar gibi graflar da doğrusal olmayan veri yapıları grubuna girerler. Bağlı listeler ve ağaçlar grafların özel örneklerindendir. Bir grafta düğümler dairelerle, ayrıtlar da çizgilerle gösterilir.

$V = \{v_0, v_1, v_2, v_3, v_4, \ldots . d_n - 1, v_n\}$ Tepeler (*vertices*)
$E = \{e_0, e_1, e_2, e_3, e_4, \ldots . e_m - 1, e_m\}$ Ayrıtlar (*edges*)
$G = \{V, E\}$ Graf (*graph*)

Bu anlatılanlardan sonra bilimsel olarak grafı şu şekilde tanımlayacağız; Bir G grafı, tepeler olarak adlandırılan boş olmayan bir $V(G)$ sonlu nesneler kümesi ile G'nin farklı tepe çiftlerinin düzensiz sıralanışı olan ve V'nin tepelerini birleştiren 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 $V\ G = v_1, v_2, \ldots, v_n$ 'nin eleman sayısına G'nin sıralanışı (*order*) denir ve $|V\ G| = n$ olarak gösterilir. Diğer taraftan ayrıtlar kümesi $E\ G = e_0, e_1, \ldots, e_m$ '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 ağırlık almasına göre **ağırlıklı graflar** veya **ağırlıksız graflar** isimleri verilebilir.

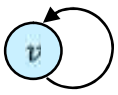
Terminoloji, Temel Tanımlar ve Kavramlar

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

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

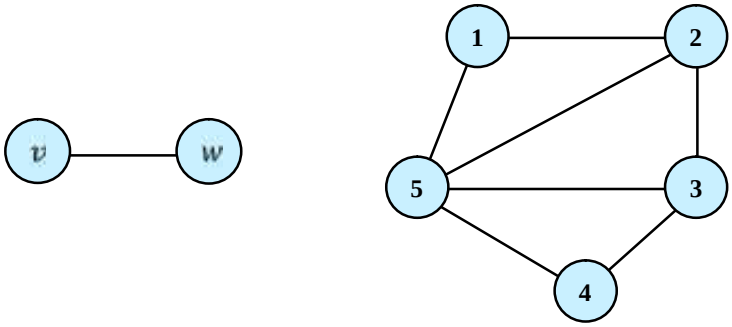
Şekil 6.2 Sıralı tepe çifti.

Self Ayrıt (*Döngü –Loop*): (v, v) şeklinde gösterilen ve bir tepeyi kendine bağlayan ayrıttır.



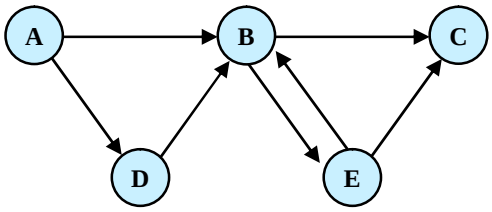
Şekil 6.3 Self ayrıt.

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



Şekil 6.4 Sırasız çift ve yönsüz graf.

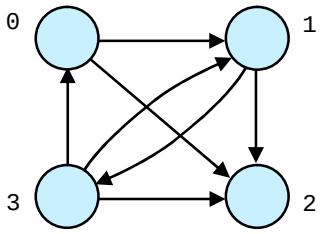
Yönlü Graf (*Directed Graph, Digraph*): Tüm ayrıtları yönlü olan grafa yönlü graf adı verilir. Yönlü grafta bir tepe çifti arasında ters yönlerde olmak üzere en fazla iki ayrıt olabilir. e sıralı bir çifttir. Her e ayrıtı bazı v tepesinden diğer bazı w tepesine yönlendirilmiştir. Yönsüz graflar, yönlü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



Şekil 6.5 Yönlü graf.

Yönlü graf tek yönde haraketli olduğu halde yönsüz graf her iki yönde de hareket eder,

Komşu Tepeler (*Adjacent*) : Aralarında doğrudan bağlantı (*ayrıt*) bulunan v ve w tepeleri komşudur. Diğer tepe çiftleri komşu değildir. e ayrıtı hem v hem de w tepeleriyle bitişiktir (*incident*) denilir.



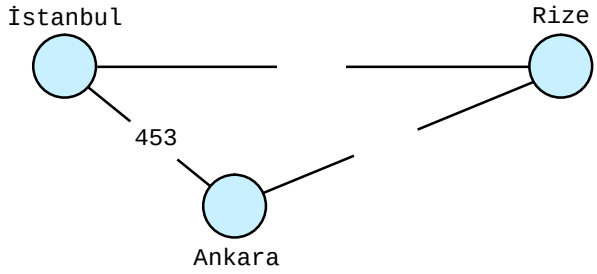
Tepe	Komşuları (*Adjacents*)
0	{1,2}
1	{2,3}
2	{}
3	{0,1,2}

Şekil 6.6 Komşuluk tablosu (*adjacency table*).

Komşuluk ve Bitişiklik: Bir G grafı komşuluk ilişkisiyle gösteriliyorsa G_{vv} = v_i, v_j … bitişiklik ilişkisiyle gösteriliyorsa G_{ve} = v_i, e_j … şeklinde yazılır. (G_{vv}: G_{tepe_tepe}, G_{ve}: $G_{tepe\_ayrıt}$)

Ağırlıklı Graf (*Weighted Graph*): Her ayrıta bir ağırlık (*weight*) veya maliyet (*cost*) değerinin atandığı graftır. Örneğin Şekil 6.5'teki graf ağırlıklı graftır.

Yol (*Path*): $G(V, E)$ grafında i_1 ve i_k tepeleri arasında $P = i_1, i_2, \ldots, i_k$ şeklinde belirtilen tepelerin bir dizisidir (E'de, $1 <= j < k$ olmak üzere, her j için $(i_j, i_j + 1)$ şeklinde gösterilen bir ayrıt varsa). Eğer graf yönlü ise yolun yönü ayrıtlar ile hizalanmalıdır. Şekil 6.7'deki grafta İstanbul'dan Ankara'ya doğrudan veya Rize'den geçerek gidilebilir.



Şekil 6.7 Ağırlıklı graf.

Yol için gevşek bağlı "*weakly connected*" denir. Herhangi bir köşeden istediğimize gidiyorsak, bu sıkı bağlıdır "*strongly connected*".

Basit Yol (*Simple Path*): Tüm düğümlerin farklı olduğu yoldur.

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

Uzunluk: Bir yol üzerindeki ayrıtların sayısı o yolun uzunluğudur.

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

Yönlü graflarda iki çeşit derece vardır. Bunlar;

Giriş Dercesi (*Indegree*): Yönlü grafta, bir tepeye gelen ayrıtların sayısına *indegree* denir.

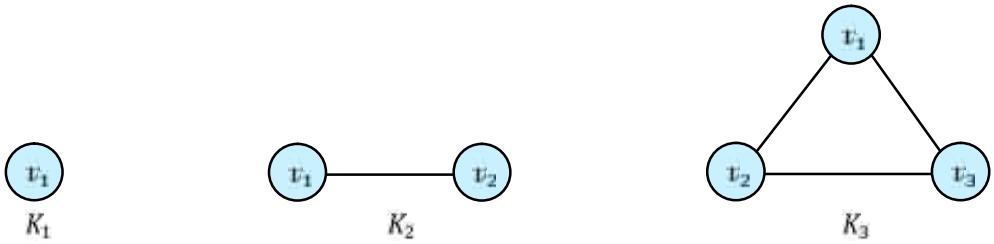
Çıkış Derecesi (*Outdegree*): Yönlü grafta, bir tepeden çıkan ayrıtların sayısına *outdegree* denilir. Örneğin Şekil 6.5'teki yönlü grafta C'nin giriş derecesi 2, çıkış derecesi ise 0'dır.

Bağlı 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 grafı G grafının alt grafıdır.

Ağaç (*Tree*): Çevre içermeyen yönsüz bağlı graftır.

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



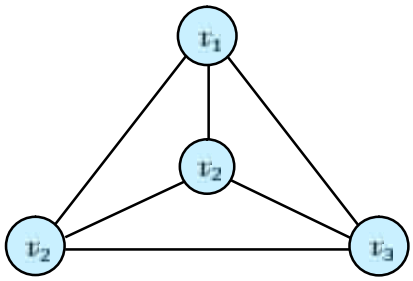
Şekil 6.8 K_1, K_2 ve K_3 tam grafları.

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

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

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

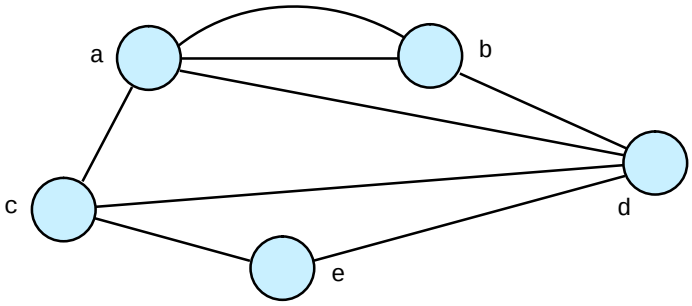
Tekerlek (*Wheel*) Graf: Bir cycle grafına ek bir tepe eklenerek oluşturulan ve eklenen yeni tepenin, diğer bütün tepelere bağlı olduğu graftır. W_n ile gösterilir.



Şekil 6.9 Tekerlek graf.

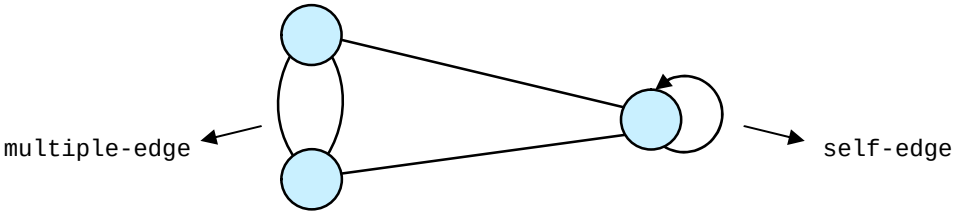
Daire veya Devir (*Cycle*): Başlangıç ve bitiş tepeleri aynı olan basit yoldur. Şekil 6.7'deki İstanbul-Rize-Ankara-İstanbul örneği.

Paralel Ayrıtlar (*Parallel Edges*): İki veya daha fazla ayrıt bir tepe çifti ile bağlanmıştır. Şekil 6.10'da a ve b iki paralel ayrıt ile birleşmiştir.



Şekil 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.



Şekil 6.11 Çoklu (*multi*) graf.

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

Forest: Bağlı olmayan ağaçlar topluluğudur.

6.2 GRAFLARIN BELLEK ÜZERİNDE TUTULMASI

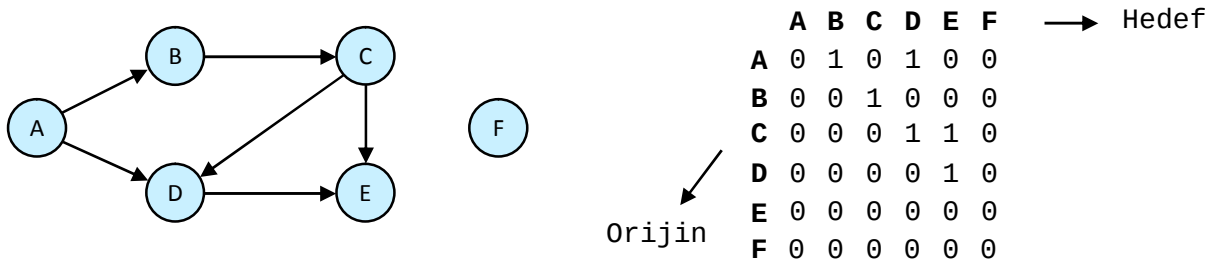
Grafların bellekte tutulması veya gösterilmesi ile ilgili birçok yöntem vardır. Bunlardan birisi, belki de en yalın olanı, doğrudan matris şeklinde tutmaktır; grafın komşuluk matrisi elde edilmişse, bu matris doğrudan programlama dilinin matris bildirim özellikleri uyarınca bildirilip kullanılabilir.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

- Komşuluk matrisi ile ardışık gösterim (*adjacency matrix representation*),
- Bağlı listeler ile bağlı 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 bir ayrıtın varlığını hızlı bir şekilde belirlemek için matrisler kullanılır. Seyrek graflar ($|E|$*'nin* $|V^2|$*'den çok daha az olduğu graflar*) için ise bağlı listeler kullanılır.

Komşuluk Matrisi (*Adjacency Matrix*)

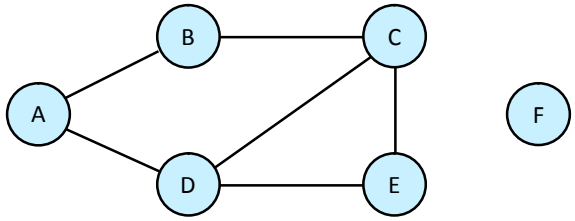
Tepelerden tepelere olan bağlantıyı `boolean` değerlerle ifade eden bir kare matristir. İki boyutlu bir dizi ile implemente edilir. $G(V, E)$, $|v|$ düğümlü bir graf ise $|v| \times |v|$ boyutunda bir matris ile bir G grafı bellek ortamında gerçekleştirilebilir. Eğer $(A, B) \in E$ ise `1`, diğer durumlarda ise `0` olarak işaretlenir. Yönsüz bir grafta matris simetrik olacağından dolayı 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.



	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	1	1	0
D	0	0	0	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

Şekil 6.12 Yönlü bir grafın komşuluk matrisi ile gösterimi.

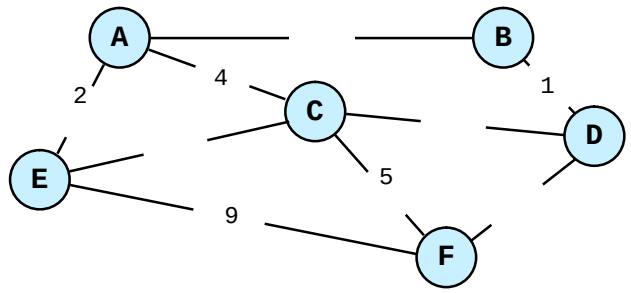
Yönlü grafların matrisi genelde simetrik olmaz fakat yönsüz graflarda bir simetriklik söz konusudur. Şimdi Şekil 6.13'te yönsüz bir grafın komşuluk matrisiyle gösterimi yer almaktadır.



	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

Şekil 6.13 Yönsüz bir grafın komşuluk matrisi ile gösterimi.

Aynı ayrıt bilgilerinin, örneğin (A, B) ve (B, A) ayrıtlarının her ikisinin birden depolanması biraz gereksiz gibi görünebilir fakat ne yazık ki bunun kolay bir yolu yoktur. Çünkü ayrıtlar yönsüzdür ve `parent` ve `child` kavramları da yoktur. Şekil 6.14'te ağırlıklı ancak yönlendirilmemiş bir grafın komşuluk matrisi gösterilmiştir.



	A	B	C	D	E	F
A	∞	8	4	∞	2	∞
B	8	∞	∞	1	∞	∞
C	4	∞	∞	7	6	5
D	∞	1	7	∞	∞	3
E	2	∞	6	∞	∞	9
F	∞	∞	5	3	9	∞

Şekil 6.14 Yönlendirilmemiş ağırlıklı bir grafın komşuluk matrisi ile gösterimi.

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

Örnek 6.1: Komşuluk matrisi ile temsil edilen yönlü bir grafta bir düğümün giriş ve çıkış derecelerini bulan fonksiyonu yazalım. Grafta 6 adet tepe olduğu varsayılsın. Bir düğümün derecesini bulmak için ilgili satır ya da sütundaki 1'ler sayılmalıdır ($O(n)$). Satırdaki 1'ler `outdegree`, sütundakiler ise `indegree` değerini verir. Bu ikisinin toplamı ise o düğümün derecesidir ($O(2n)$).

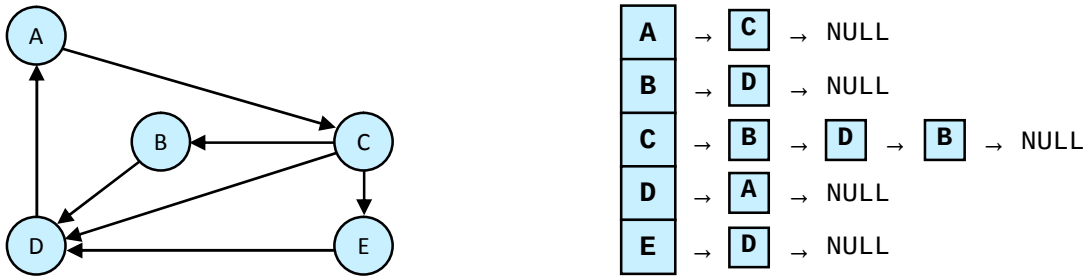
```
#define tepe_sayisi 6 // tepe sayısının 6 olduğu varsayılıyor
/* Yönlü bir grafta giriş derecesini veren fonksiyon */
int indegree(int a[][tepe_sayisi], int v) {
    int i, degree = 0;
    for(i = 0; i < tepe_sayisi; i++)
        degree += a[i][v]; // v sütunundaki 1'ler toplanıyor
    return degree;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

Çıkış 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
}
```

Komşuluk Listesi (*Adjacency List*)

Her v tepesi kendinden çıkan ayrıtları gösteren bir bağlı listeye sahiptir ve her bir tepe için komşu tepelerin listesi tutulur. Bellek gereksinimi $O(|V| + |E|)$ → dizi boyutu + ayrıt sayısı şeklindedir. Komşuluk listesi seyrek graflar için hem zaman hem de bellek açısından daha verimlidir fakat tam bir graf veya yoğun graflar için verim daha azdır.



Şekil 6.15 Yönlü bir grafın komşuluk listesi ile gösterimi.

Komşuluk listesi için veri yapısı aşağıdaki gibi tanımlanabilir;

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

Komşuluk Matrisleri ve Komşuluk Listelerinin Avantajları-Dezavantajları

Komşuluk matrisi

- Çok fazla alana ihtiyaç duyar. Daha az hafıza gereksinimi için seyrek matris teknikleri kullanılmalıdır.
- Herhangi iki düğümün komşu olup olmadığına çok kısa sürede karar verilebilir.

Komşuluk listesi

- Bir düğümün tüm komşularına hızlı bir şekilde ulaşılır.
- Daha az alana ihtiyaç duyar.
- Oluşturulması matrise göre daha zor olabilir.

Kaynaklar	**:**	Veri Yapıları ve Algoritmalar, *Rıfat Çölkesen* C ile Veri Yapıları, *Prof. Dr. İbrahim Akman*
E-mail	**:**	hakankutucu@karabuk.edu.tr
Web	**:**	http://web.karabuk.edu.tr/hakankutucu/BLM227notes.htm