DİZİLER, YIĞITLAR, KUYRUKLAR VE BAĞLI LİSTELER

Bu bölümde ele alınacak veri yapıları, diziler (tek boyutlu, çok boyutlu), yığıtlar, kuyruklar ve bağlı listelerdir. Bu veri yapıları en çok kullanılan veri yapıları olduğundan, bu bölümde detaylarının verilmesine ihtiyaç duyulmuştur.

17.1. Diziler (Arrays)

Diziler, tipleri homojen olan birden fazla elemandan oluşan veri grubudur. Diziler tek boyutlu, iki boyutlu, üç boyutlu, vb. şeklinde tanımlanabilirler. tek boyutlu diziler matematikte tanımlanan vektörlere benzerler. Bir vektördeki her elemanın tipi aynıdır. Nasıl ki matematikte tanımlanan bir vektör, kullanılmadan önce boyutu belirleniyorsa, tek boyutlu dizilerde aynı şekilde programın icra edilmesinden önce boyutu tanımlanır (dizinin kaç tane eleman içereceği belirlenir). Bundan dolayı, diziler, statik veri yapıları grubundadırlar. Örnek olarak A bir tek boyutlu dizi olsun ve elemanları tamsayılardan oluşsun ve aynı zamanda n tane elemanı olsun.

A=<100, 0, 5, ..., 6, 90>

şeklinde gösterilir veya A[1..n] şeklinde gösterilir ve her elemanın tipi belirtilir. Matematikte A dizisi

A_{1xn}

vektörü şeklinde tanımlanır ve

A=[100 0 5 ... 6 90]

şeklinde gösterilir. İki boyutlu diziler ise, iki boyutlu matrislere benzerler. Tek boyutlu dizilerde olduğu gibi elemanların tipi aynıdır.

Bunun anlamı her elemanın değer alacağı küme aynıdır. Bir dizinin elemanları için tanım kümesi tektir. Matematikte tanımlanan

$$B=\begin{matrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nm} \end{matrix}$$

matrisi dizi olarak B[1..n, 1..m] şeklinde veya

$B=<\{b_{11}, b_{12}, ..., b_{1m}\}, \{b_{21}, b_{22}, ..., b_{2m}\},...,\{b_{n1}, b_{n2}, ..., b_{nm}\}>$

şeklinde gösterilir. Daha yüksek boyutlu diziler aynı yöntem ile tanımlanırlar.

Her dizinin bir tanım kümesi vardır ve dizinin her elemanı bu kümeden değer alır. Bu tanım kümesinin elemanları basit olabileceği gibi kompleks de olabilir. Örnek olarak eğer bir okulun bir sınıfındaki öğrencilerin bir derste aldığı notlar bilgisayar ortamında saklanmak istenirse, bir öğrenciye ait kullanılacak olan bütün bilgiler dizinin bir elemanını oluştururlar. Öğrencinin adı, soyadı, numarası, dersin adı ve aldığı not yazılacaksa, dizinin her elemanı

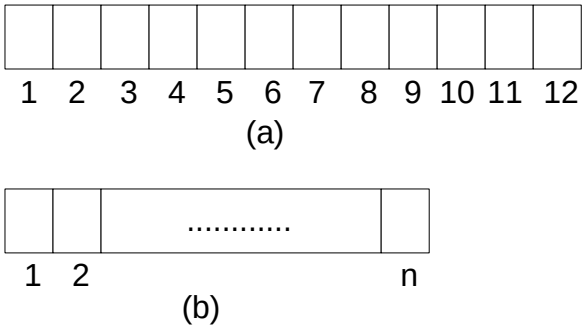
Adı : karakter dizisi
soyadı : karakter dizisi
numara : pozitif tamsayı
ders adı : karakter dizisi
not : pozitif tamsayı

şeklinde olur. Diziler genelde diğer veri yapılarında kullanılan bir veri yapısıdır. Diziler statik veriler olarak da isimlendirilebilirler.

Statik veriler üzerinde arama yapma işleminin anlatılabilmesi için verinin nasıl bir yapıya sahip olduğunun verilmesi gerekir. İlk olarak

tek boyutlu bir statik dizi verilebilir. Örnek olarak Pascal dilinde bir tek boyutlu bir dizi tanımı aşağıdaki gibi verilebilir.

$Dizi_1$: array[1..12] of $Veri\_Tipi_1$
$Dizi_2$: array[1..n] of $Veri\_Tipi_2$



Şekil 17.1. *Tek boyutlu dizi yapıları. a) 12 elemandan oluşan tek boyutlu bir dizi, b) n tane elemandan oluşan tek boyutlu bir dizi.*

Şekil 17.1 (a)' da görülen $Dizi_1$ verisinin yapısı olsun ve bu durumda her dizi elemanı $Veri\_Tipi_1$ tipinde olacaktır. Şekil 17.1 (b)' de görülen $Dizi_2$ verisinin yapısı olsun ve bu durumda her dizi elemanı $Veri\_Tipi_2$ tipinde olacaktır. Her iki dizi tanımında da bu veri boyutları program çalışmaya başlamadan önce açık olarak tanımlanmalıdır. $Dizi_2$ üzerinde arama yapmak için özelliği belli olan bir veri verilir ve bu verinin bu dizi içinde olup olmadığına bakılır. İlk olarak dizinin en küçük endeksli elemanından başlanır veya en büyük endeksli elemanından başlanır. Hangi uçtan başlandıysa, diğer uca doğru sırası ile her eleman bu verilen veri ile karşılaştırması yapılır. Eğer bu eleman dizinin içinde varsa, arama işlemi başarılı olur. Eğer yoksa, arama işlemi başarısız olur. Bu işlemi yapan algoritma Algoritma 17.1' de görülmektedir.

Algoritma 17.1. Lineer Arama(n,x)

▹ Statik ve sıralanmamış veri dizisi üzerinde arama algoritması (Lineer arama). Dizi$_2$ dizisi içinde aranacak veri x verisidir ve n dizinin boyutudur. ¬ simgesinin anlamı önüne geldiği mantıksal ifadenin değilini almaktır.

```
1-      i←1, Veri_Var←0
2-      (i<=n) ve (¬(Veri_Var)) olduğu sürece devam et
3-          eğer Dizi2[i]=x ise
4-              Veri_Var←1
5-          i←i+1
```

Bu algoritmanın analizi yapılacak olursa, kaç tane başarılı ve kaç tane başarısız arama yapılabileceği hesaplanır ve bu aramalarda kaç tane karşılaştırma yapılır, o değerler hesaplanır. Daha sonra elde edilen toplam karşılaştırma sayısı arama sayısına bölünerek bu algoritmanın mertebesi belirlenir. Algoritma analizi yapılırken en iyi ve en kötü durum analizleri de yapılır. En iyi ve en kötü durumların mertebelerinin toplamının ikiye bölünmesi ile ortalama durum mertebesi elde edilir. Fakat bu her zaman ortalama durumu vermeyebilir.

Lineer arama algoritmasının en iyi durumu, eğer arama küçük endeksten büyük endekse doğru gidiyorsa, aranan elemanın en küçük endeksli elemanda ise, bir adımda bulunur. Eğer arama en büyük endeksten küçük endekslere doğru gidiyorsa, bu durumda aranan eleman en büyük endeksli elemanda ise, bir adımda aranan değerin dizide olduğu tespit edilir.

Her başarılı arama için döngü kısmının kaç kez çalıştığı sıra ile

2, 3, 4,…., n+1,

olur. Bir aramanın başarısız olması için döngü kısmı n+1 sefer çalışır. Bu değerlerin hepsi toplanırsa

$$\sum_{i=1}^{n}(i+1)+(n+1)=\frac{n(n+5)}{2}$$

olur. n+1 tane arama yapıldığına göre elde edilen bu toplam n+1' e bölünür. Bir algoritmanın mertebesinin asimptotik olarak ifade edilmesinde sabit sayılar ihmal edilir. Lineer arama algoritmasının arama zamanı T(n) ise, bu bağıntının asimptotik sınırları

$$T(n)=\begin{cases}\Theta(1) & \text{en iyi durum}\\ \Theta(n) & \text{en kötü durum}\\ \Theta(n) & \text{ortalama durumu}\end{cases} \qquad (17.1)$$

şeklinde olur. T(n) zamanının asimptotik davranışına dikkat edilirse, iyi olmadığı rahatlıkla görülebilir. Çünkü n tane eleman için ortalama n tane arama yapılıyor ve aynı zamanda bu arama algoritmanın en kötü durumu ile aynıdır. Asimptotik notasyonlarda sabit katsayılar ihmal edilir, bunun sebebi n sonsuza giderken, bu sabit katsayıların etkisi olmayacaktır.

Lineer arama algoritması, herhangi bir dizi üzerinde yapılan bir aramadır. Dizinin elemanları sıralı olması veya olmaması herhangi bir anlam ifade etmez. Bu aramada aranacak eleman dizinin bütün elemanları ile karşılaştırılır ve bu işleme, aranan eleman dizide bulununcaya kadar veya dizide olmadığı kesinlik kazanıncaya kadar devam edilir.

Eğer dizinin elemanları sıralı ise, bu durumda mertebesi daha iyi olan bir algoritma kullanılabilmektedir. Bu algoritmanın temel mantığı aranacak elemanı dizinin ortasındaki eleman ile karşılaştırıp, aranacak eleman eğer bu elemana eşitse, amaca ulaşılmıştır. Eğer eşit değilse, bu durumda aranacak eleman dizinin hangi parçası içinde olabileceği kararı verilir. Çünkü dizinin elemanları sıralıdır. Aranan eleman dizinin ortasındaki elemandan büyükse, endeksleri ortadaki endeksten küçük olan elemanların hiçbirinde aranan eleman olmayacaktır. Aranan elemanın olabileceği kısım endeksleri ortadaki elemanın endeksinden büyük olan elemanlardan birinde olabilir. Bu şekilde bir

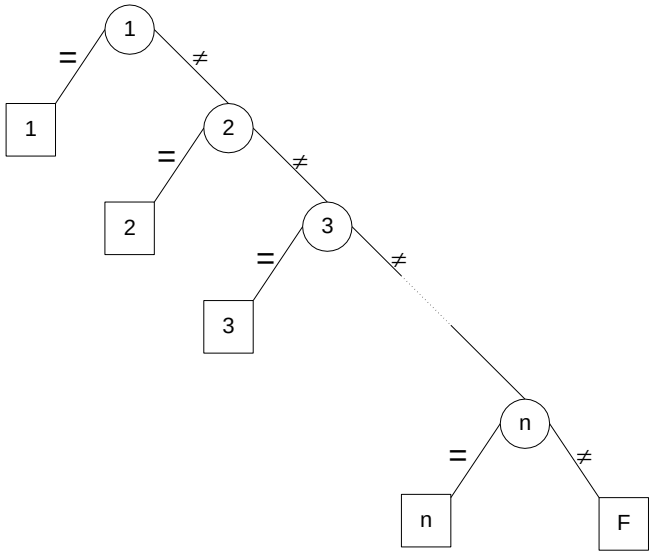
seferde dizinin yarısının bu elemanı içeremeyeceği kararı verilmiş olur. Aranan eleman ortadaki elemandan küçükse, aynı durum bunun için de geçerlidir. Bu arama yöntemine ***ikili arama*** denir. İkili algoritma Algoritma 17.2' de verilmiştir.

Algoritma 17.2. İkili Arama(Dizi$_2$,n,x,bulundu, yeri)

▷ Statik ve sıralı veri dizisi üzerinde arama algoritması (ikili arama).

```
1-      Yerel değişkenler
      alt, ust, orta : integer;
2-      ust←n, alt←1, bulundu←0
3-      (bulundu=0) ve (ust≥alt) olduğu sürece devam et
4-          orta←⌊(alt+ust)/2⌋
5-          eğer x=Dizi2[orta] ise
6-              bulundu←1
7-          değil ve eğer x<Dizi2[orta] ise
8-              ust←orta-1
9-          değilse
10-             alt←orta+1
11-     yeri←orta
```

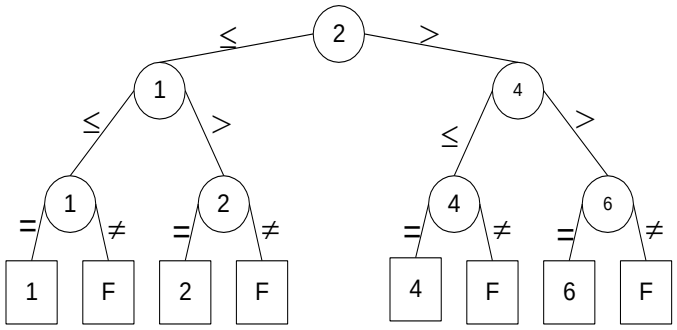
İkili aramanın mantığı veya işleyiş şekli ağaç şeklinde gösterilebilir ve bu ağaca **karşılaştırma ağacı** veya **karar ağacı** denir. Karşılaştırma ağacı nedir ve bu ağaca birkaç tane örnek verilebilir. Bir karşılaştırma ağacı, bir algoritmanın yapmış olduğu karşılaştırmaların hepsinin temsil edildiği bir ağaçtır. Örneğin, statik veri yapıları üzerinde Lineer arama algoritmasının karşılaştırma ağacı Şekil 17.2' de görülmektedir.



Şekil 17.2. *Statik veri yapıları üzerinde Lineer arama algoritmasının karşılaştırma ağacı.*

Şekil 17.2' e dikkat edilecek olursa, verilecek bir elemanın dizinin içinde olup olmadığını test etmek amacıyla dizinin her elemanı ile karşılaştırma yapılır. Bundan dolayı ağaç tek dal üzerine büyümektedir ve ağacın bir tarafı hiç yok iken, diğer tarafı çok büyüyebilir. Buradan da rahatlıkla görülebileceği gibi Lineer arama algoritması iyi bir algoritma değildir. Çünkü ağacın çok büyüyen dalından alınıp diğer taraflara verilecek şekilde bir düzenleme yapılabilir ve bu düzenleme sonucunda elde edilecek algoritmanın karşılaştırma ağacının yüksekliği Şekil 17.2' de görülen ağaçtan (aynı eleman sayıları için) daha küçük olacağından yapılacak karşılaştırma sayısı daha az olacaktır.

Statik veri yapıları üzerinde yapılan ikili arama algoritması için de karşılaştırma ağacı oluşturulabilir. Bu algoritmanın ağaçlarından biri olabileceği bir karşılaştırma ağacı Şekil 17.3' te görülmektedir.

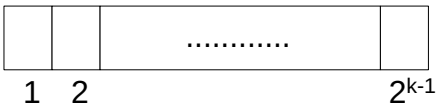


Şekil 17.3. *Statik veriler üzerinde yapılan ikili arama algoritmasının karşılaştırma ağacı.*

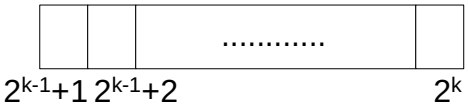
Şekil 17.3' te görülen karşılaştırma ağacı dengeli bir ağaçtır. Bazı durumlarda karşılaştırma ağacının bütün yaprakları aynı seviyede olmayabilir. En kötü olasılıkla, bir ikili arama algoritmasının karşılaştırma ağacının yaprakları bitişik iki seviyede olurlar. Her iki durumda da elde edilen ağaçların yükseklikleri (aynı veri kümesi için) lineer arama algoritmasının karşılaştırma ağacının yüksekliğinden küçüktür. Her iki algoritmanın karşılaştırma ağaçlarının yükseklikleri arasında logaritmik bir ilişki vardır.

İkili arama algoritmasının çalışma şeklinin ağaç olarak verilmesinden sonra bu algoritmanın analizi yapılabilir.

Bu algoritmanın analizini yapmak için ilk olarak $n=2^k$ kabulü yapılsın. Bu kabulün yapılması genelliliği bozmayacaktır. Algoritma çalıştığında ilk bakılacak eleman 2^{k-1} endeksli eleman olacaktır. Eğer eşitlik varsa, amaca ulaşılmıştır. Eşitlik yoksa geriye kalan ve her birinin boyutu 2^{k-1} olan dizi parçalarının hangisinde aranan elemanın olacağına karar verilir. Eğer x değişkeninin değeri $Dizi_2[2^{k-1}]$ elemanın değerinden küçükse,



x elemanı bu parçanın içinde olabilir, diğer parçanın içinde olması mümkün değildir. Eğer x değişkeninin değeri $Dizi_2[2^{k-1}]$ elemanın değerinden büyükse,



x elemanı bu parçanın içinde olabilir, diğer parçanın içinde olması olasılığı sıfırdır. Bu şekilde geriye kalan parçanın içindeki eleman sayısı 2' nin kuvveti kadar eleman kalacaktır. Bundan dolayı ortadaki elemanı bulmak için taban veya tavan fonksiyonuna ihtiyaç duyulmayacaktır. Dizi üzerinde yapılacak bölme sayısı lgn olacaktır. Bir dizi için başarılı arama sayısı n ve 1 tanede başarısız arama sayısı yapılabilir. Toplam n+1 tane arama yapılabilir. Bu algoritmada en iyi durum aramasında döngü kısmı bir sefer çalışır ve en kötü durumda döngü kısmı lgn+1 sefer çalışır. Ortalama arama zamanına bakılacak olursa, toplam arama sayısı ile arama adım sayısı çarpılır ve n değerine bölünür. Asimptotik notasyonda sabit katsayılar ihmal edildiğinden dolayı, ikili arama algoritmasının mertebesi T(n)

$$T(n) = \begin{cases} \Theta(1) & \text{en iyi durum} \\ \Theta(\lg n) & \text{en kötü durum} \\ \Theta(\lg n) & \text{ortalama durum} \end{cases} \qquad (17.2)$$

olur. Bu algoritmanın da en kötü durumu ile ortalama durumun asimptotik davranışları aynıdır. Fakat en kötü durum ve ortalama durum mertebeleri logaritmik olduğundan, lineer aramaya göre çok iyi olan bir algoritmadır. Eğer bir dizi içindeki veriler sıralı ise, her zaman ikili arama algoritmasını kullanmak sistemin performansını arttırır.

Dizi boyutu n 2' nin kuvveti olmadığı durumda tavan ve taban fonksiyonları kullanılır. Bu fonksiyonların asimptotik davranış üzerinde herhangi bir etkisinin olmadığı daha önceki bölümlerde anlatılmıştı. Bundan dolayı algoritmanın mertebesi değişmeyecektir.

İkili arama algoritmasının performansını ölçmek için, bu algoritmanın T(n) zaman bağıntısının tekrarlı bağıntısı elde edilebilir. İkili arama algoritmasının T(n) bağıntısı

$T(n)=T(\lfloor n/2 \rfloor)+\Theta(1)$

şeklinde olur. Bu tekrarlı bağıntının çözümü iteratif yapılırsa (taban fonksiyonunu ihmal ederek),

$T(n)=T(n/2)+\Theta(1)$
$=\Theta(1)+(\Theta(1)+T(n/2^2))$
$=\Theta(1)+(\Theta(1)+(\Theta(1)+T(n/2^3)))$
$=(\lg n-1)\Theta(1)+T(n/2^{\lg n})$
$=(\lg n-1)\Theta(1)+T(1)$

elde edilir. İkili arama için $T(1)=\Theta(1)$ olur, bundan dolayı

$T(n)=(\lg n-1)\Theta(1)+\Theta(1)$
$=(\lg n)\Theta(1)$
$=\Theta(\lg n)$

olur. Master yöntemi kullanılırsa, $f(n)=\Theta(1)$ ve $a=1$, $b=2$ olduğundan $n^{\log_b a}=n^{\log_2 1}=n^0=1$ olur. Bundan dolayı $f(n)=\Theta\left(n^{\log_b a}\right)$ olur. Master yönteminde bu şart sağlandığında

$T(n)=\Theta\left(n^{\log_b a}\lg n\right)$
$=\Theta(n^0\lg n)$
$=\Theta(\lg n)$

elde edilir.

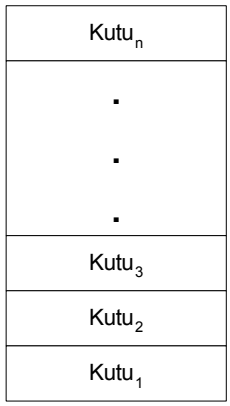
17.2. Yığıtlar (Stacks)

Çoğu insan lokantaya gitmiştir ve lokantada temiz tabaklar kullanılırken, sürekli temizlenip gelen tabak en üste konur ve tabak kullanılırken de en üstten alınır. Diğer bir deyişle en son gelen en önce işleme tabi tutulur. Bilgisayarda bu mantık çok yerde

kullanılmaktadır. Özyinelemeli olarak tanımlanan bir yordam çalışırken hafıza kullanımı bu yöntem ile ele alınır. Bir işletim sistemi süreçlerin işleme alınmasını bu mantık ile gerçekleştirir veya bir süreç devam ederken, bir kesme geldiği zaman, devam eden sürecin kesilip kesmenin devreye girmesi aynı mantık ile yapılır. Bu yöntem LIFO (Last In First Out) olarak bilinir.

LIFO mantığı ile işlem yapabilmek için kullanılan veri yapısına yığıt denir. Yığıtlar statik veriler olabileceği dinamik veriler de olabilirler. Bir yığıt statik olarak tanımlanırken, dizi olarak tanımlanabilir. S bir yığıt ise S[1 ... n] şeklinde veya S[1 ... tepe(s)] şeklinde tanımlanır ve Şekil 17.4' de yığıta bir örnek görülmektedir.

İlk olarak $Kutu_1$ yere konulur ve onun üzerine $Kutu_2$ ve onun üzerine $Kutu_3$ konulur ve bu şekilde devam edilerek en son olarak $Kutu_n$ konulur. Bu kutular alınırken ilk olarak $Kutu_n$ alınır ve ondan sonra onun altındaki alınır ve bu şekilde devam edilir. Son üç kutu kaldığında en üstte $Kutu_3$ olur ve ilk olarak o alınır ve ondan sonra $Kutu_2$ alınır ve son olarak da $Kutu_1$ alınır.



Şekil 17.4. *Kutulardan oluşan bir yığıt.*

Yığıtlara eleman eklenirken veya eleman silinirken kullanılan yordamlara özel isimler verilmiştir. Bir yığıta eleman ekleme yordamı PUSH olarak bilinir ve bu yordam yığıtı ve eklenecek elemanın

değerini parametre olarak alır. Yığıta eleman ekleyebilmek için yığıtın dolu olmaması gerekir. Yığıtın dolu olup olmadığını kontrol eden yordam Algoritma 17.3' de görülmektedir.

Algoritma 17.3. DOLU(S):Boolean

▷ yığıtın endeksi **endeks** ve yığıtın
▷ eleman sayısı **üst** olsun.

```
eğer endeks=üst ise
        DOLU ← 1
aksi durumda DOLU ← 0
```

PUSH yordamı Algoritma 17.4' de görülmektedir.

Algoritma 17.4. PUSH(S,x)

▷ x, S yığıtına eklenecek değerdir.

```
eğer DOLU(S) ise
  yığıt dolu ve eleman eklenemez
aksi durumda
        endeks ← endeks+1
        S[endeks] ← x
```

Yığıttan bir elemanın silinmesi işlemi ise POP olarak isimlendirilir ve yığıttan bir elemanın silinebilmesi için yığıtın boş olmaması gerekmektedir. Yığıtın boş olup olmadığını kontrol eden algoritma BOŞ algoritmasıdır ve Algoritma 17.5' te görülmektedir.

Algoritma 17.5. BOŞ(S) : Boolean

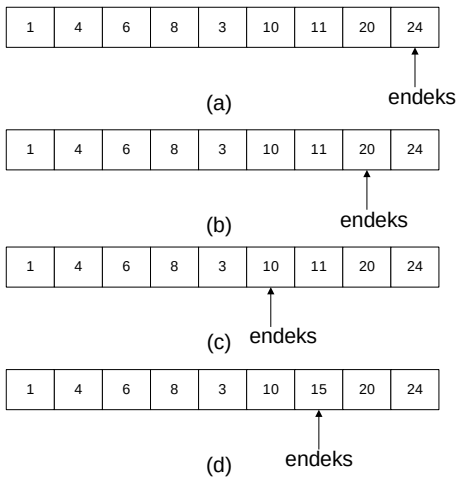
```
eğer endeks=0 ise
        BOŞ ← 1
aksi durumda BOŞ ← 0
```

Yığıttan eleman silme yordamı POP Algoritma 17.6' te görülmektedir.

Algoritma 17.6. POP(S)

```
eğer endeks=0 ise
        Yığıt boş ve eleman silinemez
aksi durumda
        endeks ← endeks-1
        S[endeks+1] döndür
```

Yığıtlar, matematiksel ifadeler için kullanılan **Polish notasyonun** hesaplanmasında ve verilen bir karakter dizisinin tersten yazılması için de kullanılabilmektedirler.



Şekil 17.5. *Bir yığıt uygulaması.*

Şekil 17.5' de bir yığıtın yığıt işlemleri sonucunda değişik durumları görülmektedir. Şekil 17.5 (a)' da dolu bir yığıt görülmektedir. POP(S) işleminden sonraki durumu Şekil 17.5 (b)' de görülmektedir. POP(S) işleminin peş peşe iki kez uygulanması sonucunda yığıtın durumu Şekil 17.5 (c)' de görülmektedir. PUSH(S,15) işleminin icrası sonucunda yığıtın durumu Şekil 17.5 (d)' de görülmektedir.

Yığıtların dinamik veriler uygulamaları da yapılabilmektedir.

17.3. Kuyruklar (Queues)

Kuyruklar, isminden de anlaşılacağı gibi veriler bir kuyruğa girmişler gibi ve bu kuyruktaki kurallar bu verilere uygulanır. Kuyruk kuralı, ilk gelen ilk çıkar şeklinde bir kuralı vardır.

Günümüzde sıkça gördüğümüz ATM kuyrukları, hastane kuyrukları, öğrenciler üniversite sınavına girmek için form alma ve form teslim etme eylemleri için de kuyruğa girerler. Kuyruktaki insanlardan ilk gelen, ilk olarak işi yapar ve gider. Aynı mantık kuyruğa yerleştirilmiş verilere uygulanır. Bu mantık FIFO (First In First Out) olarak bilinir.

Kuyruğa eleman eklemek için kuyruğun dolu olmaması gerekir ve kuyruğa eleman eklemek için 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 silmek içinde kuyruğun boş olmaması gerekir. Kuyruktan eleman silmek için de ilk önce kuyruğun boş olup olmadığı kontrol edilir ve kuyruk boş değilse, eleman silme işlemi gerçekleştirilir; kuyruk boş ise eleman silme işlemi başarısız olur. Kuyruktan eleman silme işlemi DEQUEUE olarak bilinir.

Kuyruk işlemleri gerçekleştirilirken, iki tane endeks tutulur. Bunlardan biri kuyruğun baş kısmını tutar ve diğeri de son kısmını tutar. Eleman silme işlemi kuyruğun baş tarafına uygulanır ve eleman ekleme işlemi de kuyruğun son tarafına uygulanır. Baş tarafı tutan endeks **ilk** ve son tarafını tutan endekste **son** olsun.

İlk olarak kuyruk oluşturulduğu zaman **ilk**←0 ve **son**←1 olarak ayarlanır. Bazı durumlarda **ilk** ve **son** endekslere aynı değer atanır. Bazı durumlarda kuyrukta bulunan eleman sayısını tutan **sayı** diye bir endeks tutulur. Eleman sayısını tutan endeks tutulduğunda, kuyruk ilk oluşturulduğu zaman **sayı**←0 olarak ayarlanır. Kuyruğun boyutu n olmak üzere kuyruğa eleman ekleme algoritması Algoritma 17.7' te görülmektedir.

Algoritma 17.7. ENQUEUE(Q,x)

```
1- eğer sayı=n ise
          kuyruk dolu
  aksi durumda
          Q[son]←x
2- eğer son=n ise
          son←1
  aksi durumda
          son←son+1
```

Kuyruktan eleman silme yordamı DEQUEU işlemi Algoritma 17.8' da görülmektedir.

Algoritma 17.8. DEQUEUE(Q)

```
1-        eğer sayı=0 ise
           kuyruk boş
      değilse
          Q[ilk] döndür
2-        eğer ilk=n ise
             ilk←1
      değilse
             ilk←ilk+1
```

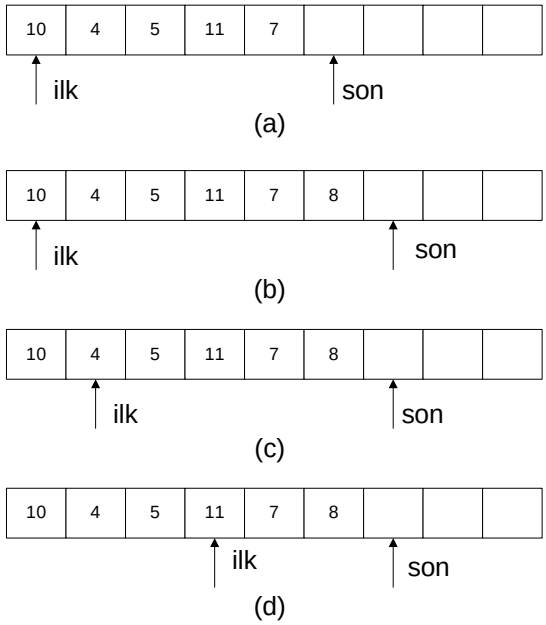
Şekil 17.6' te kuyruk işlemlerinin bir örnek üzerindeki gösterimi görülmektedir. Şekil 17.6 (a)' da kuyruğun 5 elemanı dolu şekli görülmektedir. ENQUEUE(Q,8) işlemi uygulandıktan sonra kuyruğun

almış olduğu durum Şekil 17.6 (b)' de görülmektedir. Kuyruğa DEQUEUE(Q) işlemi uygulandıktan sonra kuyruğun almış olduğu durum Şekil 17.6 (c)' de görülmektedir.

DEQUEUE(Q)
DEQUEUE(Q)

işlemleri ard arda uygulandıktan sonra kuyruğun almış olduğu durum Şekil 17.6 (d)' de görülmektedir.

Kuyrukların dinamik verilerle uygulanma şekli de vardır.



Şekil 5.6. *Kuyruk işlemlerinin kuyruğu nasıl değiştirdiği.*

17.4. Bağlı Listeler (Linked Lists)

Statik veri yapılarına örnek olarak diziler, statik yığıtlar, statik kuyruklar verildi.

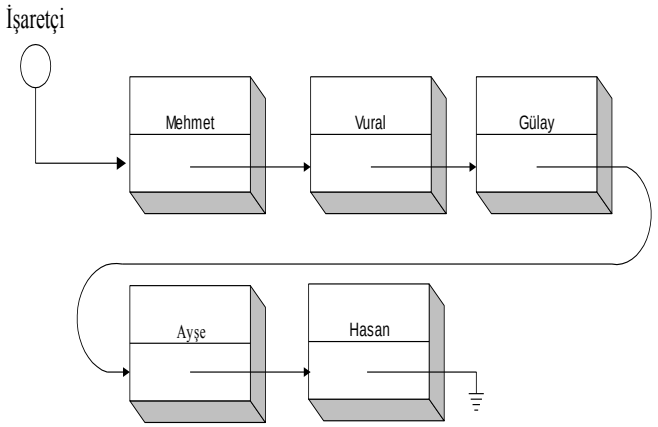
Bağlı listelerde, veriler belli bir sırada yerleştirilirler ve o ana kadar kaç tane veri geldiyse, bağlı listenin boyutu odur. Programın icrası

sırasında bağlı listenin boyutu değişebilmektedir. Bundan dolayı bağlı listelere dinamik veri yapıları denir. Dinamik veri yapıları hafızanın etkin bir şekilde kullanılması istendiği durumlarda mutlaka kullanılmalıdırlar.

Bağlı listelerin tanımlanması için veri tiplerinden işaretçi (referans) kullanılır. İşaretçi bir veri tipi olup hafıza hücrelerinin adresini tutan değişkenlerdir. Bu değişkenler hafıza adresi tuttuğundan dolayı, başka bir değişkenin bulunduğu hafıza adresini tuttuğundan dolayı bu yolla değişkenler birbirine bağlanır. Bir değişkenden yola çıkarak diğer değişkenlerin değerleri elde edilebilir.

İlk olarak bağlı listenin yapısını ve bir bağlı liste için veri yapısı

```
Type
   İşaretçi=↑düğüm;
   düğüm=record
            bilgi:string[20];
            bağlantı:İşaretçi;
   end;
```



Şekil 17.7. *Bir bağlı liste örneği.*

şeklinde tanımlanabilir. Bu tanımlamada Türkçe alfabede olup İngiliz alfabesinde olmayan bazı harfler kullanılmıştır. Bir derleyicide

bunlara hata verilir, fakat burada bu konuya dikkat edilmeyecektir. Şekil 17.7' de yukarıda yapılan tanımlamaya uygun bir bağlı liste görülmektedir ve bu listede beş tane eleman vardır

Şekil 17.7' de görüldüğü gibi bağlı liste üzerinde hareket tek yönlüdür. Bundan dolayı bir yönden hareket edilince geriye dönebilmek veya geride kalan kısımları kaybetmemek için bağlı listenin ilk elemanını tutan bir işaretçi tutulur. Başka bir yöntem olarak da çift işaretçinin kullanılmasıdır. İşaretçilerden biri baştan sona doğru giderken, diğeri sondan başa doğru gelir. Çift işaretçi ile oluşturulan bağlı listelerin üzerinde işlem yapmak için bir tane işaretçi kullanılması yeterli olur. Çünkü üzerinde bulunulan elemandan başa doğru veya sona doğru hareket etmek mümkündür; hiçbir eleman ulaşılamaz değildir. Diğer bir bağlı liste yönteminde ise, liste halka şeklinde tutulur ve tek işaretçi kullanılır. Bu listelerde de ulaşılamaz eleman olması mümkün değildir.

Bağlı liste içinde verilen bir değerin olup olmadığını kontrol etmek için bağlı listede arama yapmak gerekir. Bir bağlı listede bir elemanı arama yordamı Algoritma 17.9' de görülmektedir. L bir işaretçi olup L bağlı listesinin ilk elemanına işaret eden bir işaretçidir.

Algoritma 17.9. Arama(L,x)

```
1-      q←L
2-      q≠NIL ve q.anahtar≠x oldukça
            q←q.bağlantı
3-      q döndür.
```

q.**bağlantı** işlemi q işaretçisini bağlı liste içerisindeki bir sonraki elemanın üzerine taşır. q.**anahtar**, L listesinde elemanın bilgi kısmının bulunduğu değişkendir. Bu algoritmanın analizi daha sonraki bölümlerde yapılacaktır.

L bağlı listesine bir eleman eklemek, hafıza el verdiği sürece mümkündür. Çünkü bağlı listenin boyutu o anda kullanılan hafızanın

boyutu ile sınırlıdır. L bağlı listesine eleman ekleme algoritması Algoritma 17.10' da görülmektedir.

Algoritma 17.10. Ekleme(L,x)

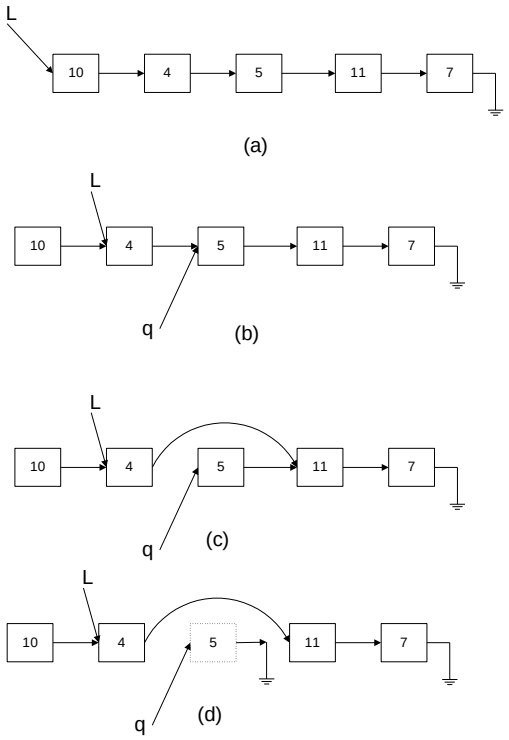
```
1-      L≠NIL do oldukça
           L←L.bağlantı
2-      L.bağlantı←yer ayır
3-      L←L.bağlantı
4-      L.anahtar←x
```

Bir bağlı listede eleman silmek için ilk önce o elemanın listede olup olmadığı kontrol edilir ve eğer silinecek eleman listede varsa, bu eleman listeden silinir ve eleman listede yoksa, silme işlemi başarısız olur. Bir bağlı listede eleman silme algoritması Algoritma 17.11' de görülmektedir.

Algoritma 17.11. Silme(L,x)

```
1-      eğer L.anahtar=x ise
          q←L
          L←L.bağlantı
          q.bağlantı←NIL
          q işaretçisini hafızadan sil.
2-      L.bağlantı.anahtar≠x ve L≠NIL oldukça
L←L.bağlantı
3-      eğer L≠NIL ise
q←L.bağlantı
L.bağlantı←L.bağlantı.bağlantı
q.bağlantı←NIL
          q işaretçisini hafızadan sil
```

Silme algoritması içerisinde aynı zamanda arama işlemi gerçekleştirilmektedir. Şekil 17.8' de x=5 elemanın silinmesi görülmektedir.



Şekil 17.8. *L bağlı listesinden x=5 elemanın silinmesi.*

Şu ana kadar anlatılan bağlı listeler en basit bağlı listeler olup her elemanın sadece bir tane işaretçi elemanı var ve elemanın geriye kalan kısmı basit veri tipindedir. Bazı durumlarda her elemanın içeriği bu kadar basit olmayabilir. Bazı durumlarda elemanın geriye kalan kısmı bir dizi olabileceği gibi bir başka bağlı liste de olabilir. Bu durumda **genel bağlı liste** denilen yeni bir kavram ortaya çıkar.

Bir bağlı listenin elemanları α_1, α_2, ..., α_n, $n \geq 0$, ise, bu bağlı liste $A=(\alpha_1, \alpha_2, ..., \alpha_n)$ şeklinde gösterilir. Genel bağlı listenin tanımı Tanım 17.1' de olduğu gibidir.

Tanım 5.1

α_i bir atom veya liste olmak üzere bir genel bağlı liste, A, sonlu eleman serisinden oluşur. A genel bağlı listesinin elemanı olan α_i atom değilse, α_i alt liste denir.

Bir genel bağlı liste $A=(\alpha_1,\alpha_2 , ...,\alpha_n)$ şeklinde gösterilir. $(\alpha_1,\alpha_2 , ...,\alpha_n)$ dizisinin ismi A ve uzunluğu n' dir. A genel bağlı listesinin başı α_1 ve kuyruğu α_n atomudur. Bir genel bağlı liste aşağıdaki durumlardan biri şeklinde olur.

(i) D=()
(ii) A=(a, (b,c))
(iii) B=(A,A,())
(iv) C=(α, C)

(i) seçeneğinde boş bir liste görülmektedir. (ii) seçeneğinde ise iki elemanlı bir liste görülmektedir ve bu elemanlardan biri atom iken diğeri ise bir listedir. (iii) seçeneğinde ise üç tane elemanı olan bir liste görülmektedir ve üç elemanı da liste olan bu listenin son elemanı boş bir listedir. (iv) seçeneğinde ise bir listenin özyinelemeli bir tanımı görülmektedir ve

C=(α, (α, (α, ..., (α,C) ...)))

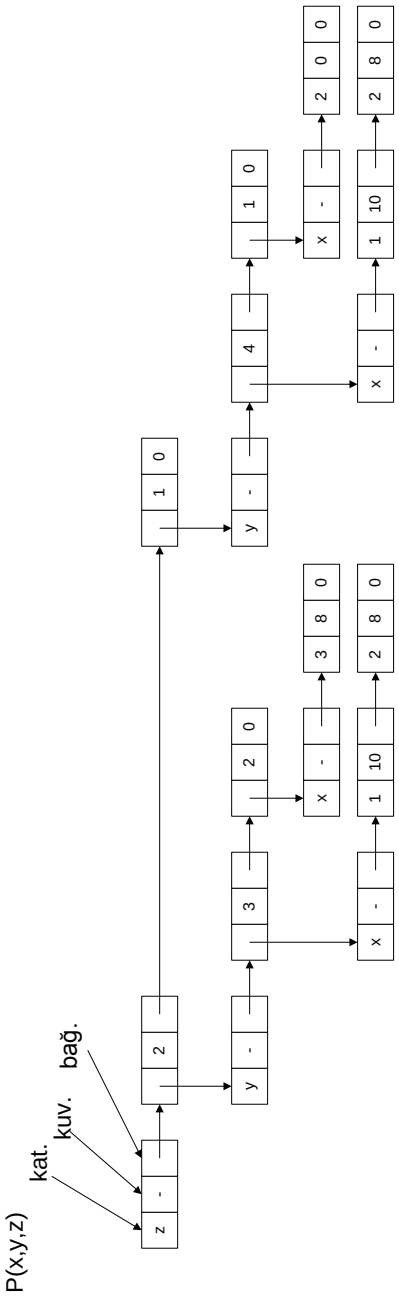
şeklinde bir listedir. (ii), (iii), (iv) seçeneğinde verilen bağlı liste için aşağıda **baş** ve **kuyruk** işlemler tanımları verilmiştir.

baş(A)='a' ve **kuyruk**(A)=(b,c)
baş(B)=A ve **kuyruk**(B)=(A,())
baş(**kuyruk**(B))=A ve
kuyruk(**kuyruk**(B))=(())

olur. Özyinelemeli ve paylaşmalı olmayan genel bağlı listeler vardır. Örneğin P(x,y,z) polinomunun bağlı liste şeklinde ifade edilmesi Şekil 17.9' da görülmektedir.

$$P(x,y,z)=x^{10}y^3z^2+2x^8y^3z^2+3x^8y^2z^2+x^4y^4z+6x^3y^4z+2yz$$
$$=z^2(x^{10}y^3+2x^8y^3+3x^8y^2)+z(x^4y^4+6x^3y^4+2y)$$
$$=z^2(y^3(x^{10}+2x^8)+3x^8y^2)+z(y^4(x^4+6x^3)+2y)$$

P(x,y,z) polinomunun paranteze alınmış şekli ile ilk sıradaki elemanlara z değişkeninin kuvveti ve katsayısı yazılır. Kuvvetler sabit



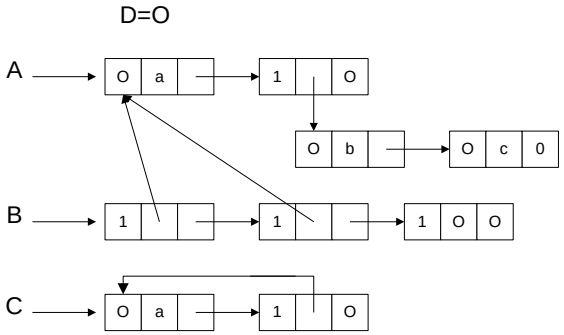
Şekil 17.9. *P(x,y,z) polinomunun genel bağlı liste şeklinde ifade edilişi.*

sayı olduklarından dolayı doğrudan yazılmıştır ve katsayıları Q(x,y) şeklinde bir polinom olduğundan dolayı bu katsayılar yerine bir listeye bağlantı eklenir. Q(x,y) polinomları

$Q_1(x,y)=y^3(x^{10}+2x^8)$
$Q_2(x,y)=3x^8y^2$
$Q_3(x,y)=y^4(x^4+6x^3)$
$Q_4(x,y)=2y$

şeklinde oldukları P(x,y,z) polinomu z parantezine alındıktan sonra görülmektedir. Her $Q_i(x,y)$, $1 \le i \le 4$, polinomu y parantezine alındıktan sonra son sırada yer alan x değişkeni için olan bağlı listeler görülmektedir ve orta sıra ise y değişkeni için tanımlanan bağlı listedir.

Daha önce verilen D, A, B ve C bağlı listeleri için elde edilen şekiller Şekil 17.10' da görülmektedir.



Şekil 17.10. *D, A, B ve C listelerin grafiksel gösterimi.*

Bağlı listelerde bir listeden bir kopyasını oluşturmak kolaydır. Fakat genel bağlı listeden bir kopya oluşturmak daha zordur, çünkü genel şeklinin nasıl olacağı ezbere bilinmez. Bundan dolayı bağlantıların dikkatli bir şekilde takip edilmesi gerekir.

Özyinelemeli olarak tanımlanan veriler üzerinde işlem yapacak olan algoritmanın tanımlanması kolay olur ve özyinelemeli olarak tanımlanır. Özyinelemeli olan bir algoritmanın her zaman iteratif şekli

tanımlanabilmektedir. Algoritma 17.12' de bir genel bağlı listenin kopyasını çıkaran bir algoritma görülmektedir. Algoritmada kullanılan listenin her elemanın üç alanı vardır ve ilk alanı o düğüme bağlı bir alt liste olup olmadığını gösteren TAG alanı var ve bu alan eğer o elemana bağlı alt liste varsa 1, yoksa 0 değerini alır. İkinci alan ise veri alanı ve son alan ise bağlantı alanıdır.

Algoritma 5.12. Kopya(L)

```
1-      işaretçi←NIL
2-      eğer L≠NIL ise
           q←Veri(L)
        değilse
           q←Kopya(Veri(L))
3-      r←Kopya(Bağlantı(L))
4-      DüğümAl(işaretçi)
5-      Veri(işaretçi)←q
6-      Bağlantı(işaretçi)←r
7-      TAG(işaretçi)←TAG(L)
8-      işaretçi döndür.
```

Algoritma 17.13. Denk(L_1,L_2)

```
1-      cevap←0
2-      Durum
: L1=NIL ve L2=NIL
    cevap←1
: L1≠NIL ve L2≠NIL
   eğer TAG(L1)=TAG(L2) ise
     eğer TAG(L1)=NIL ise
       cevap←Veri(L1)=Veri(L2)
     değilse
        cevap←Denk(Veri(L1),Veri(L2))
3- eğer cevap ise
   cevap←Denk(Bağlantı(L1),Bağlantı(L2))
```

Diğer önemli bir işlemde verilen iki genel bağlı listenin denk olup olmadığını kontrol eden işlemdir. Algoritma 17.13' de verilen iki bağlı listenin denk olup olmadığını kontrol eden algoritma görülmektedir.

Diğer önemli bir işlemde verilen bir genel bağlı listenin derinliğini hesaplayan fonksiyondur. Boş bir bağlı listenin derinliği sıfırdır ve genel formülü, listenin derinliği D(L) olmak üzere

$$D(L)=\begin{cases}0, & L=NIL\\ 1+\max(D(x_1),\ldots,D(x_n)) & L=(x_1,\ldots,x_n)\end{cases}$$

şeklinde olur. Listenin derinliğini hesaplayan algoritma Algoritma 17.14' de görülmektedir.

Algoritma 17.14. Derin(L)

```
1-      max←0
2-      eğer L=NIL ise
        max döndür
3-      işaretçi←L
4-      işaretçi≠NIL oldukça
          eğer TAG(işaretçi)=0 ise
            cevap←0
          değilse
            cevap←Derin(Veri(işaretçi))
          eğer cevap>max ise
            max←cevap
          işaretçi←Bağlantı(L)
5-      (max+1) döndür.
```

Genel bağlı listeler üzerinde yapılacak diğer önemli bir işlemde oluşturulmuş olan bir listenin hafızadan silinmesidir. Çünkü listenin yaşamı dolduysa, hafızadan silinip kaplamış olduğu hafızanın işletim sistemine devredilmesi gerekmektedir. Bir listenin hafızadan silinmesi

için her işaretçiyi işaret eden bir REF(L) alanı olduğu kabul edilsin. Eğer L listesini işaret eden k tane işaretçi varsa, bu durumda REF(L)=k olur. Verilen L genel bağlı listesini hafızadan silen algoritması Algoritma 17.15' te görülmektedir.

Algoritma 17.15. Sil(L)

```
1-      REF(L)←REF(L)-1
2-      eğer REF(L)≠0 ise
          bitir
3-      q←L
4-      Bağlantı(q)≠NIL oldukça
          q←Bağlantı(q)
          eğer TAG(q)=1 ise
            Sil(Veri(q))
5-      Bağlantı(q)←AV
6-      AV←L
```

Genel bağlı listeler üzerinde tanımlanmış bir çok işlem verilebilir; fakat burada şu ana kadar verilmiş olan algoritmalar bu kitabın kapsamı açısından yeterlidir.