

AĞAÇ (TREE) VERİ YAPISI

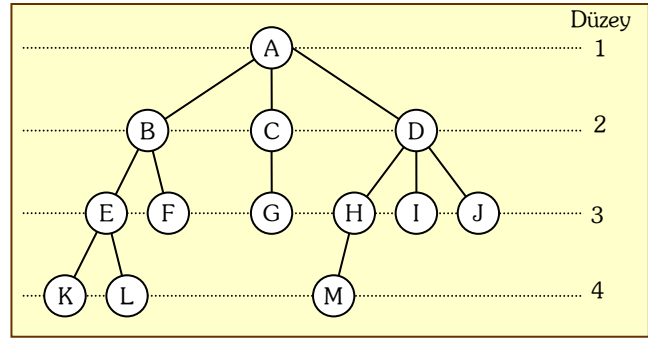
Arama, sıralama, söz dizim, veri sıkıştırma, çözümleme (*syntax analysis*), kod optimizasyonu (*code optimization*) ve derleyici gerçekleştirimindeki ara süreçler gibi çok çeşitli amaçlar için kullanılan ağaç veri yapısı, bilgisayar bilimlerinde önemli yer tutar. Ağaç veri yapısı, günlük yaşamda da karşılaştığımız bir yapıdır. Bir insanın/bitkinin soy ağacı, hiyerarşik bir yönetimdeki ilişkiler ağaç veri yapısı kullanılarak kolayca grafiksel olarak tanımlanabilir.

Ağaç (*tree*) veri yapısı, **çizge** (*graph*) veri yapısının bir alt kümesidir. Bir çizgenin ağaç olabilmesi için her bir düğümün yalnızca tek bir atası olmalıdır; diğer deyişle ağaçta her bir düğümün tek bir babası olur, çizgede ise birden çok olabilir.

TERMINOLOJİ;

Yandaki ağaç ile ilgilenelim: B; E ve F düğümlerinin babasıdır. E ve F düğümleri, B düğümünün çocuklarıdır. Aynı babanın çocukları, **kardeş** (*sibling*) düğümlerdir. Terminoloji **büyükbaba-torun** şeklinde diğer kuşaklara genişletilebilir. D, M'nin büyükbabası; E, F, G, H, I ve J düğümleri A'nın torunlarıdır.

A **kök** (*root*) düğümdür. Bir düğümün (*node*) alt ağaçlarının (*subtrees*) sayısı söz konusu düğümün **derecesini** (*degree*) verir. Örneğin A düğümünün derecesi 3'tür. F düğümünün derecesi 0'dır. Bir ağacın **derecesi** ya da **derinliği** (*depth*) ise ağaçtaki düğümlerin derecelerinin maksimumuna eşittir. Derecesi 0 olan bir düğüm **yaprak** (*leaf*) yada **uç** (*terminal*) düğüm olarak isimlendirilir. Örneğin K, L, F, G, M, I, J. Alt ağaçları bulunan bir düğüm alt ağaçlarının köklerinin **babasıdır** (*parent*). Söz konusu düğümler ise baba konumundaki düğümün **çocuklarıdır** (*child*). Bir düğümün **ataları** (*ancestor*) kökten bu düğüme kadar olan **iz** (*path*) üzerindeki tüm düğümlerdir. A, D, H düğümleri M'nin atalarıdır. E, F, K ve L düğümleri B'nin torunlarıdır.



İKİLİ AĞAÇ (BINARY TREE)

Yalnızca **sol alt ağacı** (*left subtree*) ve/veya **sağ alt ağacı** (*right subtree*) olan ya da hiç çocuğu olmayan düğümlerin oluşturduğu ağaca **ikili ağaç** adı verilir. İkili ağaçta düğümlerin dereceleri 0, 1 ya da 2 olabilir. Ağacın, çizgenin bir alt kümesi olduğunu söylemiştik. İkili ağaç da ağacın bir alt kümesidir.

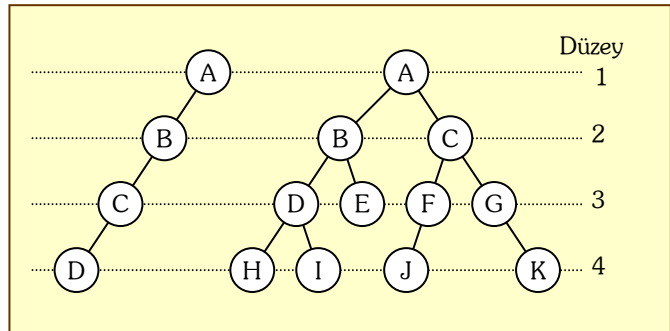
Lemma (*doğruluğu hemen görülebilen, kolayca kanıtlanabilen teorem; teoremin yumuşatılmış hali*)

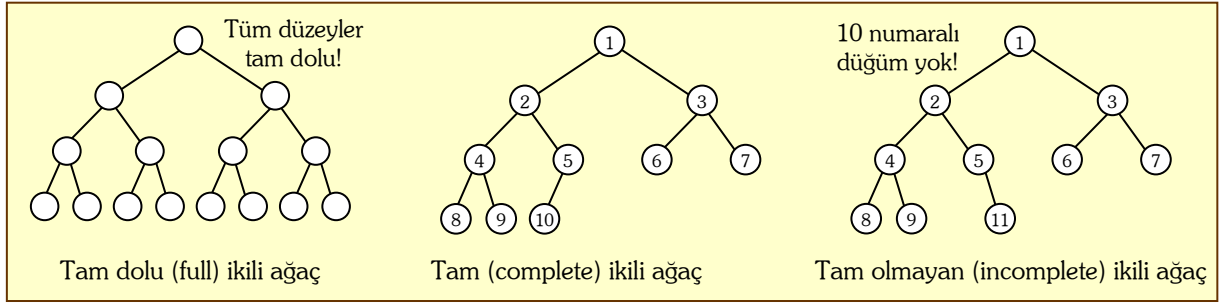
- İkili bir ağaçta i . düzeydeki düğümlerin maksimum sayısı 2^{i-1} dir. ($i \geq 1$)
- k derinliğinde (*depth*) olan ikili bir ağaçta düğümlerin maksimum sayısı $2^k - 1$ dir. ($k \geq 1$)

Lemma'nın İspatı:

i) Tümevarımdan; $i=1$ için düğüm sayısı $2^0=1$ sağlanır. $(n-1)$. düzeyde 2^{n-2} düğüm varsa n . düzeyde 2^{n-2} sol çocuk + 2^{n-2} sağ çocuk = 2^{n-1} düğüm olur.

$$\begin{aligned} \text{ii)} \quad \sum_{i=1}^k (\text{i. düzeydeki maksimum düğüm sayısı}) \\ = \sum_{i=1}^k 2^{i-1} = 2^0 + 2^1 + 2^2 + \dots + 2^{k-1} = 2^k - 1 \end{aligned}$$





TANIM: Tam dolu (full) ikili ağaç; ağacın derinliği k ise, düğümlerin sayısı $2^k - 1$ olan ikili ağaçtır.

TANIM: Tam (complete) ikili ağaç; n düğümlü olan; babaya i , sol çocuğa $2i$, sağ çocuğa $2i+1$ numarası verildiğinde 1 'den n 'e kadar numaralandırmanın yapılabildiği ağaçtır. (kökten son düğüme atlama yapılmadan numaralandırma yapılabilmeli)

İKİLİ AĞAÇ VERİ YAPISININ DİZİ KULLANILARAK GÖSTERİMİ

n düğümlü bir *tam ikili ağaç*, bir dizi üzerinde gerçekleştirilebilir. İndisi i olan bir düğüm için baba, sol çocuk ve sağ çocuk indisi aşağıdaki formüllerle hesaplanabilir. **Dizinin kullanımına -formüller gereği- 0 değil, 1 nolu indisten başlanacağına dikkat edilmelidir!**

$baba(i) = \lfloor i/2 \rfloor$ Eğer $i=1$ ise söz konusu düğüm kök düğüm olup babası yoktur.

$solcocuk(i) = 2i$ $2i > n$ ise sol çocuk kalmamıştır.

$sagcocuk(i) = 2i+1$ $2i+1 > n$ ise sağ çocuk kalmamıştır.

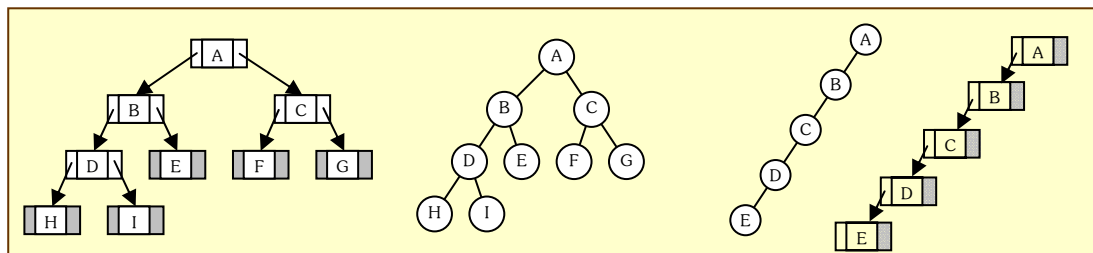
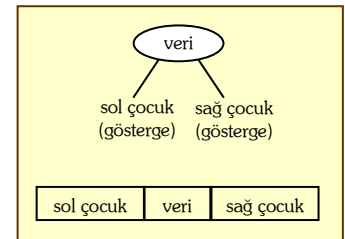
İkili ağacın dizi ile gerçekleştiriminde açılacak dizinin boyutunu belirlemede **derinlik** parametresi kullanılır. Ağacın derinliği k ise $2^k - 1$ boyutunda bir dizi açılmalıdır. Ağacımız, tam (complete) bir ikili ağaç ise (**soldaki örnek**) hemen hemen tüm dizi gözleri dolacak, bellek kaybı ya hiç olmayacak ya da çok az olacaktır.

Fakat ağaç, tam ikili ağaç olmaktan uzaklaştıkça (**sağdaki örnek**) âtil bekleyen bellek miktarı artacaktır. Örneğimizdeki sola çarpık (*skewed*) olan ikili ağaçta derinlik 5'tir. En kötü durumda $2^5 - 1 = 31$ elemanlı bir dizi açmak gerekir, fakat bu dizinin sadece 5 elemanı kullanılır.

Depolama ve erişim, bilgisayar bilimlerinin iki ana konusudur. Üstte açıklanan bellek karmaşıklığı açısından düşünüldüğünde eğer ağacımız tam ikili ağaçsa ve ekleme/çıkarma yapılmayacaksa, ağaç yapısını dizi ile gerçekleştirmek uygun olur. Örneğin bir ağ üzerindeki bilgisayarlar sabitse, ilerde bilgisayar eklenip çıkartılmayacaksa ikili ağaç yapısını dizi ile kurabiliriz.

İKİLİ AĞAÇ VERİ YAPISININ BAĞLAÇ KULLANILARAK GÖSTERİMİ

```
1: typedef struct agac *agac_gost;
2: typedef struct agac {
3:     agac_gost solcocuk;
4:     char veri;
5:     agac_gost sagcocuk;
6: };
```



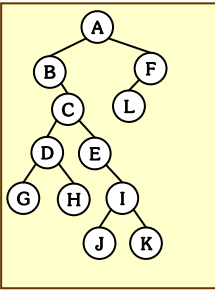
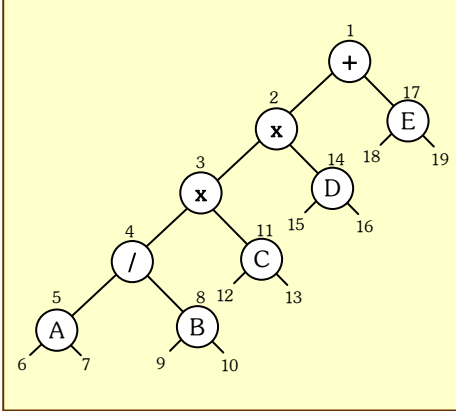
NULL göstergeler koyu renkli dikdörtgenle gösterilmiştir.

İKİLİ AĞAÇ DOLAŞIMLARI (BINARY TREE TRAVERSALS)

İkili ağaç üzerinde birçok işlem yapılabilir. Bunlardan en çok kullanılanı çeşitli kriterlere göre düğümleri yalnızca bir kez ziyaret ederek tüm ağacı dolaşmaktır. Bir ağacı dolaşırken bir düğüme ilişkin olarak *veri (i)*, *sol alt ağaç (ii)* ve *sağ alt ağaç (iii)* söz konusu olduğundan dolayı dolaşma sırası 6 farklı biçimde olabilir.

- 1- Sol – **Veri** – Sağ (*inorder* = *kök ortada*)
- 2- Sol – Sağ – **Veri** (*postorder* = *kök sonda*)
- 3- **Veri** – Sol – Sağ (*preorder* = *kök başta*)

- 4- Sağ – **Veri** – Sol
- 5- Sağ – Sol – **Veri**
- 6- **Veri** – Sağ – Sol



Kök-ortada(*infix*): (((A/B)*C)*D)+E)

Kök-önde (*prefix*): +**/ABCDE

Soldaki örnek ağacı üç farklı şekilde dolaştığımızda şu sıralamaları elde ederiz:

inorder: BGDHCEIKALF

postorder: GHDJKIECBFLA

preorder: ABCDGHEIJKFL

Gösterge	Veri Alanı	İşlev
1	+	
2	x	
3	x	
4	/	
5	A	
6	NULL	
5	A	printf
7	NULL	
4	/	printf
8	B	
9	NULL	
8	B	printf
10	NULL	
3	x	printf
11	C	
12	NULL	
11	C	printf
13	NULL	
12	x	printf
14	D	
15	NULL	
14	D	printf
15	NULL	
1	+	printf
17	E	
18	NULL	
17	E	printf
19	NULL	
Kök ortada dolaşım		

İKİLİ AĞAÇ DOLAŞMA YORDAMLARI - ÖZYİNELİ

NOT: Ağaç göstergesi NULL gelebilir, buna hiç dikkat etmiyorsunuz!

```

1: void kok_ortada_dolas( agac_gost L ){
2:   if(L){
3:     kok_ortada_dolas( L->solcocuk );
4:     printf("%c", L->veri);
5:     kok_ortada_dolas( L->sagcocuk );
6:   }
7: }

```

```

1: void kok_önde_dolas( agac_gost L ){
2:   if(L){
3:     printf("%c", L->veri);
4:     kok_önde_dolas( L->solcocuk );
5:     kok_önde_dolas( L->sagcocuk );
6:   }
7: }

```

```

1: void kok_sonda_dolas( agac_gost L ){
2:   if(L){
3:     kok_sonda_dolas( L->solcocuk );
4:     kok_sonda_dolas( L->sagcocuk );
5:     printf("%c", L->veri);
6:   }
7: }

```

NOT: Bu algoritmayı hâlâ anlamadıysanız, ağaç şeklindeki *stack frame*'i çizmeniz anlamınıza yardımcı olacaktır.

Özyineli çağrıların olduğu satırları NULL ile çağrılmaktan kurtararak algoritmamızı daha da optimize edebiliriz. Bunun için örneğin, soldaki gibi yazdığımız 4 ve 5. satırları sağdaki gibi değiştirebiliriz;

```

4: kok_önde_dolas( L->solcocuk );
5: kok_önde_dolas( L->sagcocuk );

```

```

4: if(L->solcocuk) kok_önde_dolas( L->solcocuk );
5: if(L->sagcocuk) kok_önde_dolas( L->sagcocuk );

```

Geçmişteki tüm sorularımı internete koysam, bu sizi yanıltır. Naim Süleymanoğlu'nu halter kaldırırken 100 kere seyretemsem, yine de ben o halteri kaldıramam. 'Bu nasıl yazıldı' sorusuna yoğunlaşın. Bunu siz çözmelisiniz, kendiniz anlamalısınız. Üzerinde çalışmazsanız, 4. sınıfa gelseniz bile özyineli (recursive=rekürsif) kod yazamazsınız.

Duyarsam unuturum, görürsem hatırlarım, yaparsam öğrenirim. [Çin Atasözü]

Algoritmayı yazdıktan sonra peşini bırakmıyoruz. Bilgisayar bilimlerinde algoritmanın analizini yapmak zorundayız. Yazdığımız yordamlarda (düğüm sayısı n olmak üzere) maliyet $2n = O(n)$ 'dir, değişmez. (bir gitti bir geldi, elemanların üstünden 2'ser kez geçti; toplam $2n$) Bellek karmaşıklığı ise derinlikle ilişkilidir $\rightarrow O(\lg n)$.

Sızın 'Hocam, özyineli algoritmalar çok bellek yer' tarzındaki iddialarınız, özyineli kod yazmamak için kaçamaktır. İşte bakın, bellek karmaşıklığı $O(\lg n)$ oldu. Özyineli algoritmalar, karmaşık problemlerin üstesinden hızlıca gelmek için çözüm sunar.

İKİLİ AĞAÇ DOLAŞMA YORDAMLARI - İTERATİF

Her özyineli algoritma döngüsel hale getirilebilir. Bu, iki şekilde yapılabilir. **Birincisi;** özyineli algoritmanın davranışını döngüsel şekilde ifade ederek olur. **İkincisi;** tıpkı işletim sisteminin özyineli yordam işletilirken yaptığı gibi, dönüş adreslerini bir *stack*'e atarak olur.

Biz, algoritmamızın davranışını dikkatlice tetkik edelim, örneğin **inorder** dolaşım üzerinde duralım. Kök ortada dolaşabilmek için kökteki değeri basmadan önce sol alt ağacı basmamız gerekiyor. Öyleyse düğümleri bir sıraya koyduğumuzda son eklediğimize önce erişmek isteyeceğiz. **Ya hu bu bir stack yapısı! Ne kadar estetik anlattık!** Öyleyse bir yığıt kullanacağız. Bellek karmaşıklığını artırmamak için yığita, öğeleri **kopyalamak yerine** öğelerin **adreslerini koyacağız**. Peki, bize ne kadarlık bir *stack* lazım, **yığıt boyutu** ne olmalı? Yığita aynı anda eklenebilecek en fazla eleman sayısı, **ağacın sol derinliği** kadardır.

Aşağıdaki yordam, yinelemesiz (iteratif=döngüsel) bir yaklaşımla kök ortada dolaşım gerçekleştirir. Yazdırma işlemini yapan 11. satıra uygun bir koşul ifadesi koyarsak sadece seçtiğimiz düğümleri yazdıran bir yordam elde edebiliriz: sadece uç düğümler, sadece tek çocuklular, sadece sol çocuklular vs.

```
1: void kok_ortada_dolas( agac_gost L ){
2:     void yigit_ekle( int *, agac_gost, agac_gost[] );
3:     agac_gost yigit_al( int *, agac_gost[] );
4:     int ust = -1;
5:     agac_gost yigit[MAX_SIZE];
6:     for(;;){
7:         for(;L; L = L->solcocuk )    // sol düğümleri yığita ekle
8:             yigit_ekle( &ust, L, yigit );
9:         L = yigit_al( &ust, yigit ); // Yığıt boşsa NULL döner
10:        if(!L) break;
11:        printf("%c",L->veri); //if(!(L->sagcocuk)||!(L->solcocuk)) printf("%c", L->veri);
12:        L = L->sagcocuk;
13:    }
14: }
```

Her düğüm yığita bir kez eklenir, yığıttan 1 kez alınır. Her düğümden 2 kez geçildiği için (düğüm sayısı n olmak üzere) $2n$ adım vardır, **zaman karmaşıklığı $O(n)$** 'dir. Derinlik h ise en kötü durumda (ağacın çarpık [*skewed*] olması durumunda) **bellek (*space*) karmaşıklığı $O(h)$** 'tir. Fakat bellek karmaşıklığı ağacın durumuna göre $O(1)$ 'de de kalabilir.

Bizim bu dersteki amacımız, büyük otomasyon sistemleri vs. geliştirmek değil, küçük modülleri ustaca yazabilmektir. Küçük kodları defalarca farklı şekillerde yazmalısınız. Çalışmalarınızı kâğıt üzerinde yazarak yapmanız, özyineli kod yazabilme yeteneğinizi geliştirecektir.

SORU 1: Kök önde dolaşma yordamını döngüsel olarak yazınız.

SORU 2: Kök sonda dolaşma yordamını döngüsel olarak yazınız. -> Bu daha zordur.

SORU 3: Uç düğümleri soldan sağa doğru yazan özyineli yordamı yazınız.

SORU 4: Uç düğümleri sağdan sola doğru yazan özyineli yordamı yazınız.

SORU 5: Uç düğümleri soldan sağa doğru yazan döngüsel yordamı yazınız.

SORU 6: Uç düğümleri sağdan sola doğru yazan döngüsel yordamı yazınız.

SORU 7: Yalnız sol çocuğu olan düğümleri yazan özyineli yordamı yazınız.

SORU 8: Yalnız sağ çocuğu olan düğümleri yazan özyineli yordamı yazınız.

SORU 9: Her iki çocuğu da olan düğümleri yazan özyineli yordamı yazınız.

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

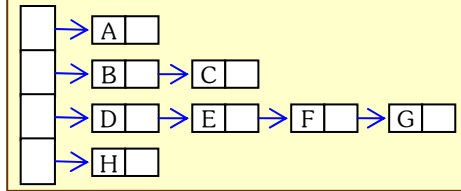
[Alan Jay Perlis]

SORU: L'nin gösterdiği bir ikili ağacın uç düğümlerini yazan yordamı yazınız.

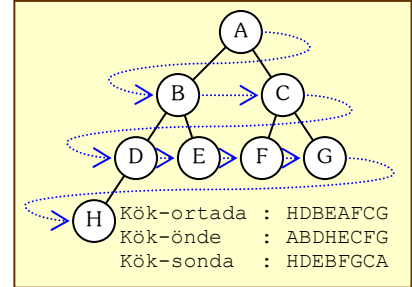
```
1: void uc_dugum_yaz( agac_gost L ){
2:     if(L){
3:         uc_dugum_yaz( L->solcocuk );
4:         if( !(L->solcocuk) && !(L->sagcocuk) )
5:             printf("%c", L->veri);
6:         uc_dugum_yaz( L->sagcocuk );
7:     }
8: }
```

SORU: İkili ağaçtaki düğümleri düzey düzey yazdıran yordamı yazınız.

I. Yöntem



Her eleman 3 kez alındığı için zaman karmaşıklığı $3n = O(n)$ olur. h, ağacın derinliği olmak üzere $\lg n \leq h < n$ ve bellek karmaşıklığı da $O(n+h)$ olur.



II. Yöntem

Ağacı düzey düzey dolaşmak için bir diğer döngüsel yöntem de kuyruk kullanmaktır.

```
1: void duzey_dolas( agac_gost L ) {
2:     void kuyruk_ekle( int, int *, agac_gost, agac_gost[] );
3:     agac_gost kuyruk_al( int *, int, agac_gost[] );
4:     int on=0, arka=0;
5:     agac_gost kuyruk[MAX_SIZE];
6:     if( !L ) return; // ağaç boş gelebilir
7:     kuyruk_ekle( on, &arka, L, kuyruk );
8:     for(;;){
9:         L = kuyruk_al( &on, arka, kuyruk );
10:        if(L){
11:            printf("%c", L->veri);
12:            if(L->solcocuk) kuyruk_ekle(on, &arka, L->solcocuk, kuyruk);
13:            if(L->sagcocuk) kuyruk_ekle(on, &arka, L->sagcocuk, kuyruk);
14:        } else break;
15:    }
16: }
```

Zaman karmaşıklığı $2n = O(n)$

Bellek karmaşıklığı $O(\lg n)$

(Derinlik $\lg n + 1$)

SORU: L ile işaret edilen ikili ağacın kopyasını çıkartan yordamı yazınız.

Önce kâğıt üzerinde benim tahtada yaptığım çalışmayı yapmalısınız. Özyineli çözüldüğünü görmelisiniz. Bu yordam *bottom-up* çalışıyor, *top-down* da yazılabilirdi. Bu yordam, ağacı kök-sonda dolaşma biçimine göre dolaşılıyor; satırların yerleri değiştirilerek kök-önde ve kök-ortada biçimine de kolayca çevrilebilir.

```
1: agac_gost agac_kopyala( agac_gost L ){
2:     agac_gost T;
3:     if(L){
4:         T=ogeal();
5:         T->solcocuk = agac_kopyala(L->solcocuk);
6:         T->sagcocuk = agac_kopyala(L->sagcocuk);
7:         T->veri = L->veri;
8:         return T;
9:     }
10:    return NULL;
11: }
```

SORU: İki ağacın eş biçimli (*isomorphic* [izomorfik]) olup olmadığını denetleyen programı yazınız.

```
1: int esitmi( agac_gost L, agac_gost T ){
2:     return ( !L && !T ) || ( L && T && ( L->veri == T->veri )
3:         && esitmi( L->solcocuk, T->solcocuk )
4:         && esitmi( L->sagcocuk, T->sagcocuk ) );
5: }
```

Eğer bizden sadece iki ağacın biçimce benzer olup olmadığının testini yapmamız isteniyorsa, veriler önemsenmiyorsa o zaman `(L->veri == T->veri)` ifadesini kaldırırız.

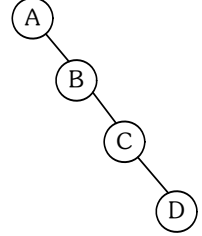
İkili ağaçların izomorfik olup olmadığını denetlemek bu kadar kolay! İki çizgenin (graf) eş biçimliliğini bulmak ise oldukça zor bir problemdir.

"Hocam, benimki çok güzel çalışıyor da geri dönmüyor. Çağırıyorum, çağırıyorum, ama dönmüyor, tıkanıp kalıyor." O zaman yine özyineli algoritma yazmayı öğrenememişiz demektir.

SORU: `esitmi()` yordamını farklı bir biçimde yazınız.

SORU: Bir ikili ağacın sağ ve sol düğümlerinin yerini değiştiren yordamı yazınız.

```
1: void cocukdegistir( agac_gost L ){
2:     agac_gost p;
3:     if(L){
4:         SWAP( L->solcocuk, L->sagcocuk, p );
5:         cocukdegistir( L->solcocuk );
6:         cocukdegistir( L->sagcocuk );
7:     }
8: }
```



Verileri değiştirmiyorum, sadece göstergeleri takas ediyorum. Böylece sağ ve sol alt ağaçlar yer değiştirmiş oluyor. Bu yordamın davranışını kâğıt üzerinde adım adım inceleyiniz. Eğer `SWAP` makrosunu özyineli çağrılarının öncesine değil de arasına ya da sonuna koysaydık ne şekilde bir davranış gözlemlerdik, tetkik ediniz.

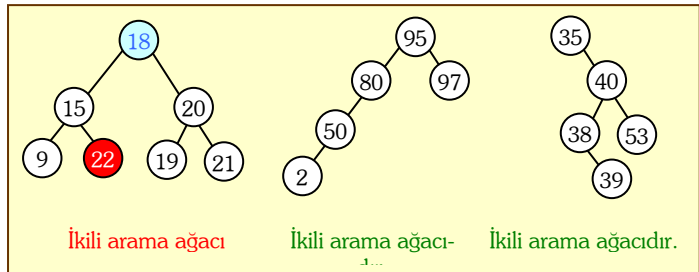
TETKİK SONUCU: Bu ağaç üzerinde aynı sonucu üretti, çünkü sağ ağaçların hepsi tek düğümlü.

İKİLİ ARAMA AĞAÇLARI (BINARY SEARCH TREE)

İkili arama ağacı da bir tür ikili ağaçtır (*ikili ağacın bir alt kümesi*). Eğer ikili arama ağacı boş değilse aşağıdaki özellikleri taşır.

1. İkili ağaçta her ögenin değeri farklıdır, birden çok öge aynı değere sahip olamaz. **Değerler tektir.** (*Unique*)
2. Boş olmayan **sol alt ağaç**taki değerler, alt ağacın kökü konumunda bulunan değerden **küçüktür**.
3. Boş olmayan **sağ alt ağaç**ta bulunan değerler, alt ağacın kökü konumundaki değerden **büyüktür**.
4. Sol ve sağ alt ağaçlar da ikili arama ağacı özelliğini taşır.

En soldaki örnekte 22 verisi içeren düğüm, 15'in sağ çocuğu olduğu için beklenen şekilde 15'ten büyüktür. Fakat 22, 18'in sol alt ağacında olduğu halde 18'den büyüktür, hâlbuki küçük olmalıydı. Bu yüzden bu ağaç, ikili arama ağacı tanımına uymaz, ikili arama ağacı özelliği taşımaz. Diğer örneklerde ise ağacın mevcut düğümleri büyüklük-küçüklük sıralamasını bozmadan yerleşmiştir.

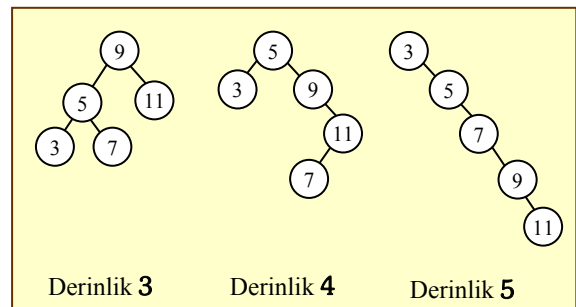


İKİLİ ARAMA AĞACINA EKLEME

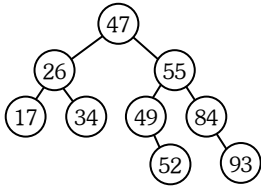
Mevcut bir ikili arama ağacına, "*ikili arama ağacı olma özellikleri*" bozulmadan her değer eklenebilir. Gelen veriye göre büyük-küçük karşılaştırmaları yapılarak uygun yer bulunup oluşturulan yeni düğüm eklenir. Siz bir değer söyleyin, ben ekleyeyim. "Hocam 15'in sağına ekleyin." Sağa eklerim, sola eklerim; sen bir değer söyle. Nereye ekleyeceğim benim bileceğim iş! 😊

SORU: 3, 5, 7, 9, 11 kümesiyle derinliği 3, 4 ve 5 olan ikili arama ağaçları çiziniz. Bu kümeyle 2 derinliğinde bir ağaç çizilebilir mi?

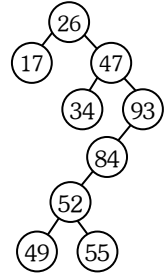
ÇÖZÜM: İstenen örnek ağaçlar yanda çizilmiştir. Bu derinliklerde farklı ağaçlar üretmek elbette mümkündür. Fakat 2 derinliğinde bir ağaç çizilemez. Çünkü lemmaya göre k derinliğindeki bir ağaçtaki maksimum düğüm sayısı $2^k - 1$ olabilir. 2 derinlikli ağaçta da en fazla $2^2 - 1 = 3$ düğüm bulunabilir, bu ağaca 5 düğüm sığmaz.



Verilerin Ağaca Eklenme Sırası Önemli midir?



İkili arama ağacını kurmada verilerin geliş biçimi (ağaca ekleniş sırası) önemlidir. Sırasıyla 47, 55, 49, 26, 34, 17, 52, 84, 93 şeklinde gelirse **soldaki** ağaçtaki gibi yerleşir. **Derinlik: 4.** Sırasıyla 26, 17, 47, 93, 84, 52, 34, 55, 49 şeklinde gelirse **sağdaki** ağaçtaki gibi yerleşir. **Derinlik: 6.** Kök ortada dolaşım sonucu ise her iki ağaç için aynı olur: 17, 26, 34, 47, 49, 52, 55, 84, 93.



Veriler sıralı gelseydi oluşacak ağaç sağa ya da sola çarpık olacaktı ve **derinliği n** olacaktı. Eğer ikili arama ağacının **derinliği $\lg(n)$** ise bu durumda **dengeli ikili arama ağacı** söz konusudur. Silme ve eklemede karmaşıklık $O(\lg n)$ olacaktır. Örneğin, $n=2.000.000$ öge için karşılaştırma sayısı $\lg n \approx 21$ olacaktır. **$O(n)$ ile $O(\lg n)$ arasındaki fark 2.000.000 ile 20 arasındaki fark gibidir.**

Dengeli ağaçlar: Veriler nasıl gelirse gelsin, ağacı dengeli hale getiren algoritmalar var. Bu algoritmalar, veri geldiğinde ağacın ağırlık noktasını anında hesaplar ve ağacı bu noktadan silkeleyerek dengeler. (*self-balancing binary search tree*) **Ör: Red-Black tree, AVL tree** .Şu an konumuz değil. Ağacın dengeli olması sizin için çok önemliyse ağacı dengeli kuran algoritmalar kullanmalısınız. Çok zor değil, kitaplar da da var, sadece biraz daha uzun. Ödev olarak sorabilirim.

Sık kullanılan düğümler: Bazı durumlarda aranan düğümler, sık kullanılan düğümler olabilir. Örneğin bir hastaneye giriş yapan hastalarla ilgili işlemler birkaç gün sürecektir, dolayısıyla aynı hastalara sürekli erişilmek istenecektir. Arama yordamının hızlı sonuç döndürebilmesi için bu düğümlerin köke yakın olması istenen bir durumdur. Ağacı, **arama frekansı** yüksek olan düğümleri köke yaklaştıracak şekilde yeniden düzenleyen algoritmalar (*Zig-zag algoritmaları*) mevcuttur.

Bunlar daha ileri konular. Şimdilik sadece gelen verilerle basitçe bir ikili ağaç kurmayı öğreneceğiz. Böylece şimdiye kadar yazdığımız yordamları çalıştırıp test etmemizi sağlayacak örnek ağaçlar oluşturabileceğiz. Şimdiye kadar arka planda bir ağaç olmadığı için kodu çalıştırarak test edemiyordunuz.

???: Ağaçta düğümlerin yerleşiminden (ağaç görüntüsünden) yola çıkarak verilerin eklenme sırası elde edilebilir mi? Gelen veriye göre, oluşan ağaç yapısı *unique* oluyor mu?

???: Kök-başa, kök-ortada, kök-sonda dolaşım sırasından ağaç görüntüsü elde edilebilir mi?

Ekleme Nasıl Yapılır?

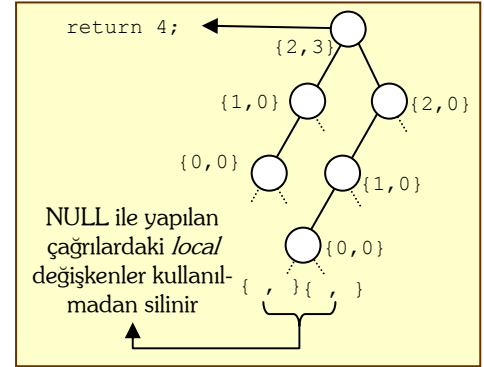
Her iki çocuğu da olan bir düğüme eklenemez. En az bir çocuğu olmayan düğüme eklenebiliyor. Gelen veri, kökten itibaren ağaçta aranır. Daha büyük değerli bir düğüme rastlanırsa sol çocuğuna, daha küçük değerli bir düğüme rastlanırsa sağ çocuğuna ilerlenir. Eğer aynı değer bulunursa ikili arama ağacının tanımına aykırı olacağından ağaca ekleme yapılamaz. Arama göstergesinin son kaldığı konuma (değeri ararken ilk rastladığı NULL göstergenin yerine) öge eklenir.

SORU: L, ikili arama ağacının kökünü işaret eden göstergedir, NULL da olabilir. İkili arama ağacına x değerini ekleyen **boolean** arama_agacina_ekle(int x, agac_gost *L) yordamını yazınız.

```
1: typedef enum { FALSE, TRUE } boolean;
2: boolean arama_agacina_ekle( int x, agac_gost *L ){
3:     agac_gost ogeal(void);
4:     agac_gost q, p, r;           // q arkadaki, p öndeki gösterge
5:     p = *L;                     // p kökü gösteriyor
6:     while( p ) {                // öndeki gösterge NULL oluncaya kadar ilerle
7:         q = p;                  // arka göstergeyi günle.
8:         if( x < p->veri ) p = p->solcocuk; // yeni elemanın yeri sol alt ağaçta mı
9:         else if( x > p->veri ) p = p->sagcocuk; // yeni eleman yeri sağ alt ağaçta mı
10:        else { printf("\nBenzer deger var."); return FALSE; }
11:    }
12:    r = ogeal();
13:    r->solcocuk = r->sagcocuk = NULL;
14:    r->veri = x;
15:    if( *L == NULL ) *L = r; // ağaç boşsa yeni öge kök yapılır
16:    else {
17:        if( x < q->veri ) q->solcocuk = r; // babanın sol çocuğu mu
18:        else q->sagcocuk = r; // yoksa sağ çocuğu mu günlenecek
19:    } return TRUE; }
```


DİKKAT: Problem yeterince küçüldü mü? Başta da NULL gelebilir!

```
1: int derinlik( agac_gost L ){
2:   /* sol ve sağ alt ağaç derinliği bulunup
3:     bu iki değerin maksimumuna 1 eklenmelidir. */
4:   int sol, sag;
5:   if ( L ){ // gösterge NULL mı
6:     // sol ve sağ alt ağacın derinliğini bul
7:     sol = derinlik( L->solcocuk );
8:     sag = derinlik( L->sagcocuk );
9:     return MAX(sol, sag) + 1;
10:  }
11:  else return 0; // uç yaprağa ulaşıldı
12: }
```



Binlerce kez özyineli kod yazdıysanız bu kodu daha profesyonelce şöyle yazarsınız:

```
1: int derinlik( agac_gost L ){
2:   return (!L) ? 0 : MAX( derinlik( L->solcocuk ), derinlik( L->sagcocuk ) )+1;
3: }
```

Ek Parametreler Koyma Mevzusu

If you have a procedure with 10 parameters, you probably missed some. [Alan Jay Perlis]

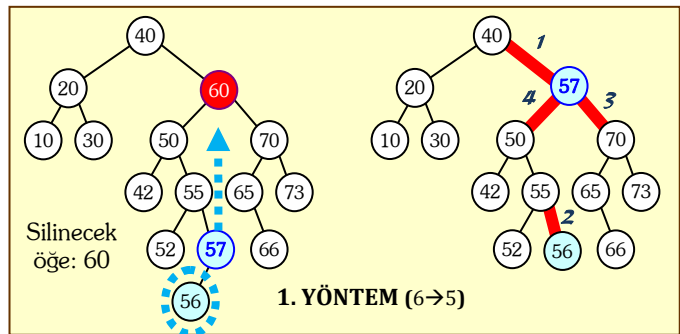
Bir öğrenci sorar: “Hocam yordamı, `int derinlik(agac_gost L, int a)` şeklinde fazladan bir parametre ekleyerek tanımlasak da `derinlik(L, 0)` şeklinde bir ilk çağrıyla çözebilir miyiz?”

Cevap: Neden fazladan parametre tanımlayıp ilk çağrıda sıfır aktarıyorsun? Aşağıda bir yerde toplayacağın dimi? Bak, ben biliyorum. Yordamı kullanan adama ne diyeceksin? —“a’ya sıfır aktarmayı sakın ha unutma!” Sıralama yordamı yazdın; 0 aktarıncı sıralıyor da, 1 aktarıncı sıralayamıyor... ☺ Olur mu? Ama şu olur; küçükten büyüğe mi, büyükten küçüğe mi sıralanacağını belirtmek için bir parametreye, -1, +1 ya da '>', '<' aktarırısın. Karakterleri sıralamak için ayrı yordam, sayıları sıralamak için ayrı yordam yazmayayım, hepsini tek yordam yapsın deyip verinin karakter mi sayı mı olduğunu bildiren bir parametre eklersin. Bu şekilde parametre sayısını artırabilirsin.

İKİLİ ARAMA AĞACINDAN ÖĞE SİLME

İkili arama ağacından düğüm silerken ağaç yapısının bozulmamasını sağlamalıyız. Çocuksuz ve tek çocuklu düğümleri silmek kolaydır, iki çocuklu bir düğümü silmek içinse farklı yöntemler vardır.

1. Yöntem: Silinecek ögenin sol alt ağacındaki en büyük değer bulunur, silinecek ögenin konumuna taşınır. *Alt ağaçtaki en büyük değeri bulmak için;* sol alt ağaçta sürekli sağ işaret alanları üzerinden ilerlenir, sağ çocuğu olmayan öge en büyüktür. Bu ögeye ait bir sol alt ağaç da mevcut olabilir. (57’ye ait 56 çocuğu vardır.) Bu sol çocuk taşınacak ögenin atasının sağ bağ alanına aktarılmalıdır. (56, 57’nin atası olan 55’in sağ bağına)



Ögeyi taşımak; iki şekilde yapılabilir. **Birincisi;** silinecek öge free edilir, gerekli bağlar güncellenerek uygun öge silinenin yerine taşınmış olur. Bu durumda -silinen ögenin sağ/sol çocuk olmasına bağlı olarak- silinenin atasına ait sağ/sol bağ bilgisi [1] taşınacak ögeyi, taşınacağın atasına ait sağ bağ bilgisi [2] taşınacağın sol çocuğunu, taşınacağın sağ [3] ve sol [4] bağları ise silinenin sağ ve sol bağlarını gösterecek şekilde güncellenmelidir. (40’ın sağ bağı 57’yi, 55’in sağ bağı 56’yı, 57’nin sol bağı 50’yi, 57’nin sağ bağı ise 70’i göstermelidir.) Görüldüğü gibi 4 adet bağ bilgisini güncellemek gerekmektedir. (kalın gösterilen bağlar)

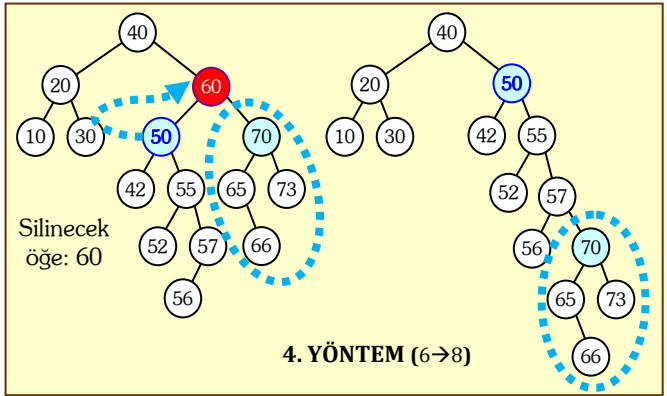
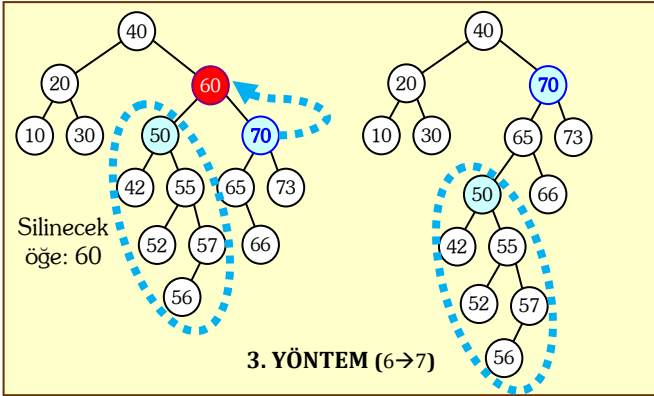
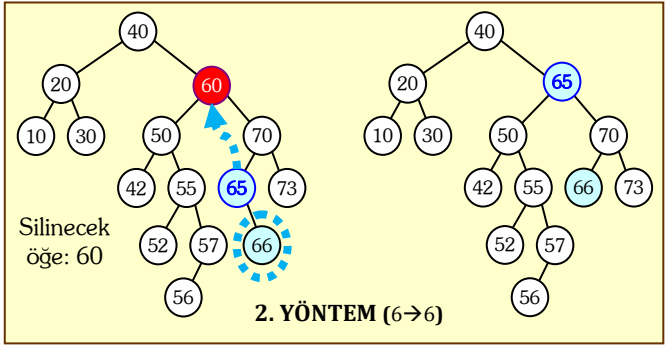
İkincisi; silinecek öge free edilmeyip verisi, taşınacak ögenin verisiyle değiştirilir. (60’ın veri alanına 57 aktarılır.) Bu durumda taşınanın atasına taşınanın sol çocuğunu sağ çocuk olarak bağlamak için bir bağ güncellemek [2] gerekir. (57’nin atası olan 55’e, 57’nin çocuğu olan 56’yı bağlamak için 55’in sağ bağ bilgisi güncellenir.) Görüldüğü gibi 1 bağ bilgisini güncellemek yeterlidir. (55’i 56’ya bağlayan bağ)

Bu arada, öge sildikçe ikili arama ağacı olma özelliğinin bozulmadığını test edip görebilirsiniz.

2. Yöntem: Silinecek ögenin sağ alt ağacındaki en küçük değer bulunur, silinecek ögenin konumuna taşınır.

3. Yöntem: Silinen ögenin konumuna silinenin sağ çocuğu kaydırılabilir.

4. Yöntem: Silinen ögenin konumuna silinenin sol çocuğu kaydırılabilir.

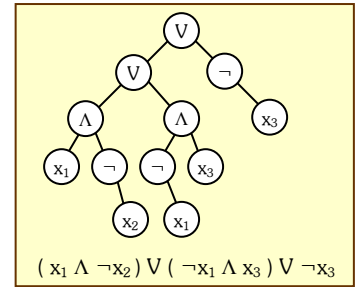


4 farklı yöntemle düğüm silindiğinde ağacın derinliğinin nasıl değiştiğine bakınız. **Sildiğimiz ögenin kökü olduğu ağacın (kökünde 60 bulunan ağaç) olabildiğince dengeli olmasını sağlayacak şekilde silmeliyiz.** (Sildiğimiz öge, ağacın diğer kısımlarında bir derinlik değişmesine sebep olamaz.) Bunun üzerinde çalışılabilir. Tüm yöntemler bütün düğümler silinerek uygulanır, derinliklerin ortalamalarına bakılarak bir sonuca varılabilir.

SATISFIABILITY PROBLEM

İkili ağaçlarla aritmetik-mantıksal ifadelerin değerlendirilmesi arasında bir ilişki vardır. x_1, x_2, \dots true ya da false değerini alabilen mantıksal değişkenler; \wedge, \vee, \neg mantıksal işleçler (and, or, not) olmak üzere mantıksal ifadeler birer önermedir (bileşik önerme). [propositional calculus] [wiki]

Satisfiability problem, değişkenlere mantıksal ifadenin sonucunu **true** yapacak şekilde bir atamanın yapıp yapılamayacağı konusunu içerir. $(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$ biçimindeki mantıksal ifadede 3 değişken olması nedeniyle, denetlenmesi gereken 2^3 olası durum vardır: {false, false, false}, {false, false, true}, ...



Problem *non-polynomial* bir problemdir. (*Computational Theory* dersini keşke almış olsaydınız. Bu derste problemleri sınıflandırıyoruz.) Yani polinomsal zamanda çözülemez. n tane değişken varsa 2^n tane durum vardır. BruteForce ile (kas kuvvetiyle, kaba kuvvetle, bilgisayarın gücüne güvenerek) çözmeye kalkarsak $O(2^n)$ zamanda çözebiliriz. İşlemci sayısını artırıp m yaparsak ve yükleri de işlemciler arasında eşit şekilde dağıtabilirsek o zaman $O(2^n/m)$ olur. Fakat yine de üstel zamanda çözüyoruz. Sezgisel olarak bir şeyler yakalayıp neleri denemek gerektiğini saptamalıyız. (*Yapay anlayış dersini almış olsaydınız; greedy algoritmaları, dinamik programlama*) Akıllıca kullanılan sezgisel bir işlev (*heuristic function*) tanımlamalıyız. 2-3 tane değişken olunca problem polinomsal zamanda çözülebilir hale geliyor. Bu konudaki sezgisel yaklaşımla ilgili çok sayıda makale var.

Daha iyi çözüm aramak: Veri yapıları derslerinde problem çözerken veri yapınızdan asla emin olmayın. Sürekli yazın, arkadaşlarınızla tartışın, “Daha iyi bir çözüm yöntemi vardır da biz bulamıyoruz.” deyin.

I haven't failed, I've found 10,000 ways that don't work. (Benjamin Franklin)

```

for( 2n sayıdaki olası tüm kombinasyonlar )
    Bir sonraki kombinasyonu elde et.
    Değişkenlerin değerlerini yerleştir.
    Kök-sonda dolaşımı ile ağacı gez. (kok_sonda_degerlendir yordamı)
        if( kök->deger ){
            printf(<kombinasyon>);
            return;
        }
}

1: typedef enum { not, and, or, true, false } logical;
2: typedef struct agac *agac_gost;
3: typedef struct agac {
4:     agac_gost solcocuk;
5:     logical veri;
6:     short int deger;
7:     agac_gost sagcocuk;
8: };
9: void kok_sonda_degerlendir( agac_gost L ){
10:    if(L){
11:        kok_sonda_degerlendir(L->solcocuk);
12:        kok_sonda_degerlendir(L->sagcocuk);
13:        switch( L->veri ){
14:            case not : L->deger = !(L->sagcocuk->deger); break;
15:            case and : L->deger = L->sagcocuk->deger && L->solcocuk->deger; break;
16:            case or : L->deger = L->sagcocuk->deger || L->solcocuk->deger; break;
17:            case true : L->deger = TRUE; break;
18:            case false: L->deger = FALSE; break;
19:        }
}

```

Circuit Satisfiability (CIRCUIT-SAT): Verilen bir devrenin çıkışını 1 yapacak bir atama yapılabilir mi?
Formül SAT

3-CNF-SAT (3-conjunctive normal form): Her alt cümle (clause) 3 değişken içeriyor;

$$(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$$

Peki, bu ağaç nasıl kurulur? Algoritmaları var, çalışma yapmak gerekir. Yazılım Mühendisliği Lab. (341-342) derslerinde 1–2 kez ödev olarak sorduk.

Algoritmamız olası tüm ihtimalleri deniyor, *brute force* yapıyor. İşlem ağacını başta bir kez kurduktan sonra (*precomputing*) her seferinde değerleri yerleştirip sonuca bakmak işimizi hızlandırıyor. **Precomputing:** Defalarca kez yapılacak bir iş için önceden bir kez yapılacak bir çalışma, hazırlanacak bir veri yapısı verimliliği son derece artırır. **Ör:** *String search/match* (en zor algoritmalarından biridir). Google’a yazdın, tak buldu! Bak, hızlı algoritma! [Knuth–Morris–Pratt](#). Lütfen algoritma analizi kitaplarından *greedy algoritmaları*, *branch and bound*, *precomputing* konularına çalışın!

OKUMA PARÇASI: Meslek Sevgisi

Fenâ fi’l-meslek olmalısınız. Mısır’da kazı çalışmalarının başına geçenler var ya! Kendisini mesleğine öylesine vermiş ki... **Adam 2009’dan besleniyor, ama 5500 sene öncesinde yaşıyor.** “*Şu firavunun karısı oraya o sebepten gömülmüş olamaz, şundandır*” diyebiliyor. Sanki o zamanda yaşıyor gibi... Bu kadar çok sevmelisiniz mesleğinizi...

Öbür adam “*Orda kara delik var.*” diyor ve ispatlamak için yıllarca oranın fotoğraflarını çekiyor. Fotoğrafın birinde yıldızın biri azıcık hareket ediyor -onun bir piksel oynaması bile çok hızlı hareket etmesi demektir-, çok seviniyor. **Adam galaksinin merkezinde yaşıyor.** Onu teleskopun başına 50 yıl koysan, koluna da serum versen; oradan hiç kalkmasa hiç bir şey demez.

Gençler buradan İstanbul’a maça gidiyorlar. 8 saat yol, yorgunluk... Üstüne 150 TL verip maça giriyorlar. Üstelik kafasına her şey gelme ihtimali var; taş, değnek... Dayak da yeyip geliyorlar, bu oluyor. Ama birisi kalkıp “*Filanca hoca gelmiş, İstanbul’da konferans verecek.*” diye 8 saatlik yola, İstanbul’a gitse; “*Vay aptal!*” denir değil mi? Bunu yapan yok. Peki, olması gereken o mu, yoksa bu mu?

Mesleğinizi çok sevmelisiniz. Siz mesleğinizi severseniz herkes sizi dinler, elinin ucuyla anlatırsan kimse dinlemez. **Kodlarda da, kodun içinde yaşıyormuş gibi ne olduğunu anlamalısın.**

Mustafa Ege

... Bir ömür mutlu olacaksanız, işinizi sevin. *Çin Atasözü*

GENELLEŞTİRİLMİŞ LİSTELER (*GENERALIZED LISTS*)

Doğrusal liste (*linear list*); n elemanı ($n \geq 0$) olan sonlu bir listedir. $A = \{ \alpha_0, \alpha_1, \dots, \alpha_{n-1} \}$ ile sembolize edilsin. A listesinde α_i ; α_{i+1} elemanından önce gelir ($0 \leq i < n-1$). Tek bağlaçlı liste, dizi vb. Bu tür bir listede öğelerin her biri atomiktir.

Tanım: $A = \{ \alpha_0, \alpha_1, \dots, \alpha_{n-1} \}$ biçiminde verilen bir listede A , listenin **adıdır**. Liste **uzunluğu** n 'dir. α_i ($0 \leq i \leq n-1$) atomik ya da bir alt liste olabiliyorsa bu tür listeye **genelleştirilmiş liste** denir. α_0 **liste başı** (*head of A*), $(\alpha_1, \alpha_2, \dots, \alpha_{n-1})$ **liste kuyruğu** (*tail of A*) olarak isimlendirilir.

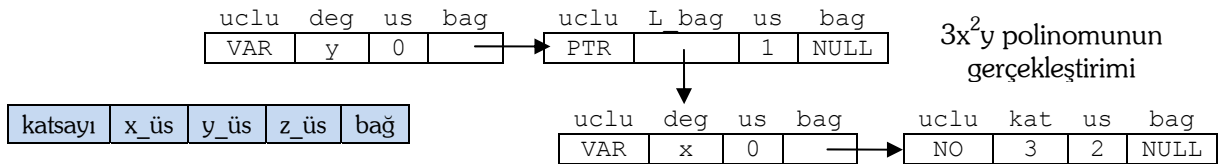
Örnekler:

$D = ()$ boş (*empty*) liste; uzunluğu 0 (sıfır)
 $A = (a, (b, c))$ uzunluğu 2, ilk elemanı atomik, 2. elemanı b ve c elemanlarını içeren bir alt liste
 $B = (A, A, ())$ paylaşımlı alt liste içeren bir liste (*list with shared sublist*); 3 elemanlı, ilk 2 elemanı A listesi, 3. elemanı boş liste
 $C = (a, C)$ özyineli liste; 2 elemanlı

Elimizde şöyle bir polinom olsun;

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

Polinomu gerçekleştirmek için ardışık yerleşim (*sequential represent*) bir çözüm olarak düşünülebilir:

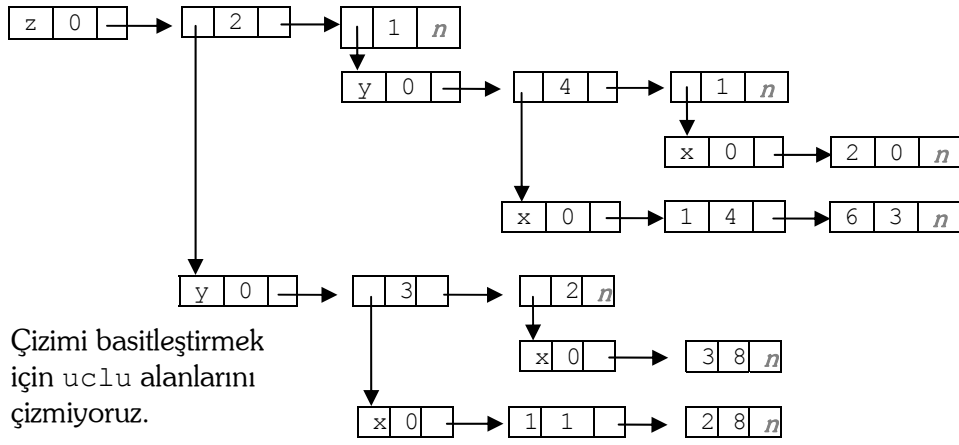


```
1: typedef enum { VAR, PTR, NO } boolean;
2: typedef struct oge *oge_gost;
3: typedef struct oge {
4:     boolean uclu;    // üçlü
5:     union{
6:         char deg;    // değişkenin adı
7:         oge_gost L_bag;
8:         int kat;      // katsayı
9:     };
10:    int us;           // üs bilgisi
11:    oge_gost bag; };
```

Eğer $uclu = "VAR"$ ise düğüm, liste başıdır. deg 'de değişken adı tutulur. us 'e sıfır yerleştirilebilir.

Eğer $uclu = "PTR"$ ise katsayı, L_bag ile işaret edilen listedir.

Eğer $uclu = "NO"$ ise katsayı tamsayı bir değerdir, kat değişkeninde tutulur. Son iki durumda us değişkeninde ilgili değişkenin üs bilgisi vardır.

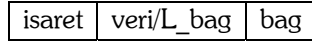


$$P(x, y, z) = ((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z$$

Polinomların genelleştirilmiş liste ile gerçekleştirimi için bu, tek bir örnek, sadece ilk örnek. Başka şekilde de gerçekleştirebilirsin. Ama bunu yapan dönüşüm yordamlarını da yazabilmelisin.

Bilgisayardaki dosya hiyerarşisi, web sitelerinde / programlarda kullanılan popup menüler de bunun gibidir. (paylaşımlı alt liste içeren liste)

Genelleştirilmiş listeler için şu yapıyı kullanmak da mümkündür:



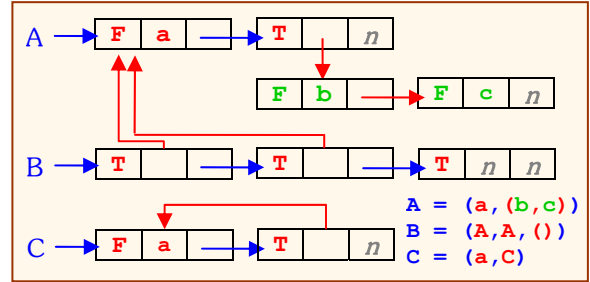
isaret = FALSE ise öge atomiktir.

```
1: typedef enum { FALSE, TRUE } boolean;
2: typedef struct oge *oge_gost;
3: typedef struct oge {
4:     boolean isaret;
5:     union{
6:         char veri;
7:         oge_gost L_bag;
8:     }
9:     oge_gost bag; };

```

Genelleştirilmiş Listelerin Avantajı

Veriyi ağaç olarak tutmak da mümkündür, ama *text*le karşılaştırılmaz. “Ben *bitmap* saklarım. Ağacın bellek görüntüsünü saklarım.” Olur, saklayabilirsin. Ama bu şekilde tutarsak daha az yer kaplıyor, bir nevi sıkıştırma yapıyoruz. Onun buna karşılık gelmesi mümkün değil. Genelleştirilmiş listelerin avantajı bu! Az yer kaplama avantajı. Bu kadar küçük bir verinin saklanması bile bilgisayar mühendisi dikkatli olmalıdır.



Bir verinin başka bir uzaya taşınması... *Z-transformation*... Çok iyi bildiğin, hâkim olduğun bir uzaya taşıyorsun. Biz de veriyi aldık, çok iyi bildiğimiz algoritma uzayına taşıdık.

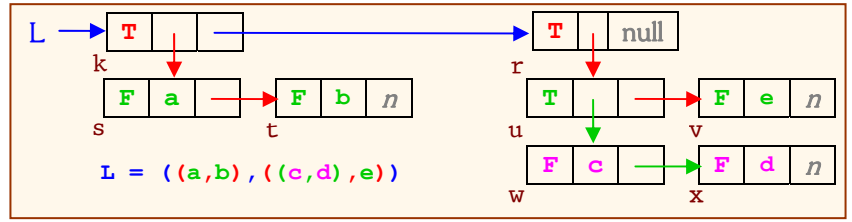
SORU: Paylaşımlı alt liste içermeyen özyinelemesiz bir listeyi kopyalayan yordamı yazınız.

```
1: oge_gost kopyala( oge_gost L ){
2:     oge_gost q = NULL;
3:     if( L ){
4:         q = ogeal();
5:         q->isaret = L->isaret; // işaret bilgisini günle
6:         if(!L->isaret) //öge atomik mi?
7:             q->veri=L->veri;
8:         else q->L_bag=kopyala(L->L_bag); // paylaşımlı alt listeyi kopyala
9:         q->bag = kopyala(L->bag); // bag alanını günle
10:    }
11:    return q;
12: }

```

B = (a, ((), b, c, (d, e, (f)), k), e)

Yandaki liste ile yordamı test ediniz. Alırken baştan sona, günlerken sondan başa... “Hocam ben özyineli yazmıycam.” diyen var mı hala? “Ben onu kuyruğa koyarım.” diyen arkadaş varsa artık bir şey diyemeyeceğim...



Özyineli bir listede bu kod, sonu gelmeyen özyineli çağrılar sebebiyle çakılırdı. Paylaşımlı listeden iki kez geçerdi. Örneğin; B=(A,A,()) listesini kopyalamaya çalıştığımızı düşünün. O yüzden “Bu yordam, paylaşımsız listeyi başarıyla kopyalar.” diyoruz.

Listeyi meydana getiren yordam, quizlik soru! Listeyi kuran yordamı nasıl yazarım, kopyalayanı nasıl yazarım diye sürekli düşünün! Ustalaştığınızda özyineli olarak yazabilirsiniz.

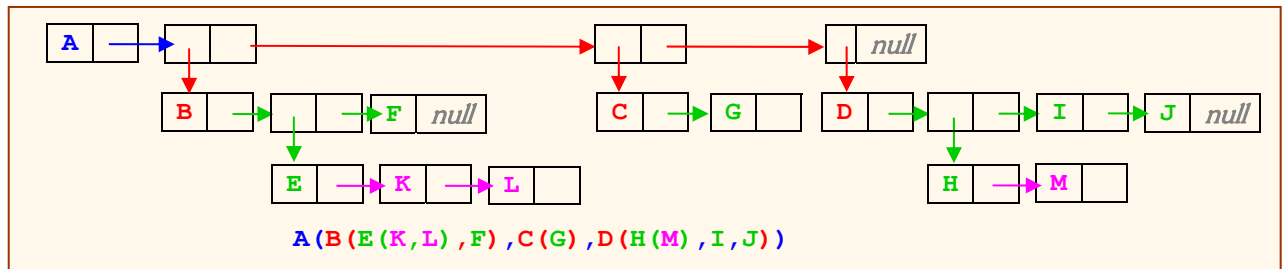
SORU: Paylaşımsız ve özyinelemesiz iki listenin eşit olup olmadığını bulan yordamı yazınız.

```

1: boolean esitmi( oge_gost T, oge_gost L ){ // T ve L özyinelemesiz listeler
2:     boolean x = TRUE;
3:     if( !T && !L ) return TRUE;
4:     if( T && L && ( T->isaret == L->isaret ) ) {
5:         if( !T->isaret ){ // öge atomik midir?
6:             if( T->veri == L->veri ) x = TRUE; else x = FALSE;
7:         } else // öge atomik değil, alt listeler eşitmi
8:             x = esitmi( T->L_bag, L->L_bag );
9:         // verilen alt listeler eşitse diğer öğeler eşitmi
10:        if( x==TRUE ) return esitmi( T->bag, L->bag );
11:    } return FALSE; }

```

Bu kod, ustaca yazılmış bir kod değil. Daha nitelikli kod yazabilirsiniz. Bir de sonsuz çağrılar sebebiyle özyineli listeler karşılaştırılmaz. **Bakın, programın direnç noktasını tespit ediyorum.** Tuşa bastın, çalıştı; yetmez! O, 100 adımdan bir tanesi... Tabii doğru çalışacak, yanlış çalışacak hali yok! Matematikte yaptığınız ispatlar gibi; “ $\sqrt{2}$ irrasyonel sayıdır, ispatlayınız.” “Hocam ne gerek var ispatlamaya? Biz zaten biliyoruz onun irrasyonel olduğunu...” demek gibidir. Hâlbuki ispat yaparken daha geniş bir bilgi sahasına hâkim olduğunu gösterirsin. Öyle olmasaydı bilgisayar bilimleri algoritmadan ibaret olurdu. “Aşağıdaki algoritma bunu bulur, yukarıdaki şunu bulur...” Hiçbir ispat yok, direnç noktası yok...



SORU: Özyinelemesiz ve paylaşımlı iki listenin eşit olup olmadığını bulan yordamı yazınız.

SORU: Özyinelemesiz ve paylaşımsız bir genelleştirilmiş listeyi dolaşp, atomik düğümlerdeki değerleri yazan `void dolas_yaz(oge_gost L)` yordamını yazın.

```

1: void dolas_yaz( oge_gost L ){
2:     if(L)
3:         if(!L->isaret)
4:             printf("%c", L->veri); // öge atomikse yaz
5:         else
6:             dolas(L->L_bag);
7:         dolas( L->bag );
8: }

```

SORU: Özyinelemesiz bir listenin derinliğini bulan yordamı özyineli olarak yazınız.

```
1: int derinlik( oge_gost L ){
2:     oge_gost p = L;
3:     int m=0; // max
4:     if(!L) return 0; // özyineli
5:     else{
6:         while(p){
7:             if(p->isaret){
8:                 int n = derinlik( p->L_bag );
9:                 if( m<n ) m=n; }
10:            p=p->bag;
11:        }
12:        return m+1; }}
```

SORU: Özyinelemesiz bir listenin derinliğini bulan yordamı döngüsel olarak yazınız.

Öğrenci Çözümü:

```
1: int derinlik( agac_gost L ){
2:     agac_gost yigit_al();
3:     void yigit_ekle(agac_gost);
4:     int max = 0;
5:     for( int current = 1, p = L ; p ; p->bag ){
6:         if( p->isaret ){ // alt listeye gir, derinlik artar
7:             yigit_ekle(p);
8:             p = p -> L_bag;
9:             current++;
10:        }
11:        if( !p->bag ){ // liste sonu, üste çık
12:            p = yigit_al();
13:            if( max < current ) max = current;
14:            current--;
15:        }
16:    }
17:    return max;
18: }
```

Özyineli Düşünme: Bu dersi alıyorsan, bu kodu *current*, *max* değişkenleri tutarak döngüsel olarak yazmayacaksın. Onu 1. sınıf öğrencisi yazar. Sen 2. sınıf öğrencisisin. **Özyineli algoritmalarda mantığı kavrayabilirseniz bu kadar kısa kodla problemi çözmek mümkün.** İteratif de çözün, öneririm. Ama global değişken kullanmak yasak! İçinde yığıt kullan, kuyruk kullan, *local* değişken kullan, ne istersen kullan. Ama **global değişken yok!** Sana bir *component* verirler. “Şunu yap.” derler. O bir *network security*’de bir parçadır ya da işletim sisteminin bir parçasıdır. Onu sen bilmezsin.

Bana kalsaydı özyineli düşünme altyapısını vermek için bu derse 3 değil 15 kredi önerirdim. Bu altyapıyı vermek için... Bizim eski bir öğrencimiz Siemens’te işe girmiş. Anlatıyor; “Bir soru vardı hocam, ODTÜ matematik mezunu birine sordum. Kalemimi bir şu yana oynattı, bir bu yana oynattı. Sonra; ‘Bu rekürsif!’ dedi.” diyor. “Biz niye bunu yapamıyoruz, hemen göremiyoruz?” demek istiyor.

Yemek yerken, yürürken, televizyon izlerken –zaten siz televizyon izlemiyorsunuzdur da- sürekli düşünmelisin. “Onu ne yaparsam ne döndürürüm?” Probleme ancak bu şekilde tüm hislerinle yönelebilirsin. Lütfen *debug* yaparak da nereye ne dönüyor, görün, özyineli algoritma yazma becerinizi geliştirin. Benim burada anlattığımı siz kod yazarken görebilmelisiniz.

Doğru çalışan kod: Bir de sadece verilen örnek üzerinde doğru çalışan kod yazıyorsunuz. Nasıl beceriyorsunuz, bilmiyorum. Yani; verilen örnek üzerinde doğru çalışan, başka örneklerde çakılan kod yazma becerisine sahipsiniz. Vallahi bravo! Hâlbuki farklı örnekler üzerinde, uç örnekler üzerinde kodunuz doğru çalışmalıdır.

Orijinal Kitap Almak: Her dersten 2–3 kitabınız olmalı. Para verip alırsanız okursunuz. Fotokopi çektiürseniz okumazsınız. Ders notlarını ders bitince atarsınız. Fakat 80–90\$ verip aldığınız kitabı atmazsınız. Bu alışkanlığı bu yaşlarda öğrenin. Son derece önemlidir. İşe girdiğinizde hocanın anlatmadığı ya da anlattığı halde sizin anlamadığınız bir konuyu oradan çalışırsınız. **Google soru gruplarını** kaçırmayın, oralardan soru çözün. Muhtemelen iş teklifi için bir değerlendirme yapıyorlar.

SORU: Paylaşımsız özyinelemesiz listenin devriğini alan yordamı yazınız. Ör: $L=(a,(b,c)) \rightarrow L=((c,b),a)$

Bu problemle gerçek yaşamda çok karşılaşırız. Ör: *Image Processing*’de sağa bakan resmi sola çevirmek. “Text’ten çevirsek?” Hayır, onu istemiyoruz. Görüldüğü gibi *Veri Yapıları 1* dersinde yazdığımız matris devriği alan yordama sadece bir ekleme yaptık. O yordamı iyi ki biliyoruz! Yoksa “Ya, bunun devriğini nasıl alırım ki?” diye düşünecektik. Liste başı değişebileceği için adresle gönderdik. 7. satırdaki `L_bag` alanının özyineli çağrılardan dönmeden önce 13. satırda `*L = q;` ifadesiyle günclendiğine dikkat edin.

```
1: void devrik_al( oge_gost *L ){
2:     oge_gost r,
3:     p=*L,
4:     q=NULL;
5:     while(p){
6:         if( p->isaret )//atomik değilse
7:             devrik_al(&p->L_bag);
8:         r=q;
9:         q=p;
10:        p=p->bag;
11:        q->bag=r;
12:    }
13:    *L = q;
14: }
```

Bir kişi *recursive* mantığı bilmiyorsa ne yapardı? Matriste mi tutardı, iç içe yığıtta mı tutardı... Kolay kolay da bir yerde tutamazdı. `devrik_al` yordamını iyi ki biliyoruz. Böyle çözmezsek *brute brute force* olurdu. Sonuçta o da devrik alıyor, ama çamları devirip devirip gidiyor.

İşe başladığınızda amirinizle aranızda şöyle bir diyalog geçmesi muhtemeldir: “Veri yapıları dersini aldın mı?” Siz de; “Evet, aldım.” diyeceksiniz. Bunun üzerine; “Ben de aldım. Şu list, tree falan değil mi?” diye soracak. “Evet; list, tree falan...” diyeceksiniz. Aynı şunun gibi; “Beyin cerrahisi? Ha, şu da-mar değil mi? Ha işte o, tamam ya!” Sanki o kadar basitti. **Veri yapıları, bilgisayar bilimlerinin bel kemiğidir. List’i tree’yi bilmek, veri yapısını öğrenmiş olmak değil. Bunu yapan yordamı yazabilmelisiniz.**

```
1: void serbest_birak( list_gost *list ){
2:     oge_gost p = *list;
3:     if( *list ){
4:         p = *list->bag;
5:         free(*list);
6:         serbest_birak(&p);
7:     }}
```

```
1: void serbest_birak( list_gost *list ){
2:     oge_gost p = *list;
3:     if( p ){
4:         *list = p->bag;
5:         free(p);
6:         serbest_birak(list);
7:     }}
```

Bu ne kodu? Ne çabuk unuttunuz ya! Final sorusu. **Tek bağlaçlı bir listenin öğelerini teker teker serbest bırakan yordam** değil mi? Soldaki kodu incelemeniz isteniyordu. *Call by reference* ile aktarıldığına göre **ana yordamdaki list içeriği** değişmeliydi. Bunu çağıran yordama `list`, `NULL` dönmüyor. Ana yordamdaki adam hala önünde bir liste görüyor. *Kodu yazdın, çok eminsin. Ama o kadar emin olma!* Soldaki gibi değil, sağdaki gibi yazılmalıdır.

OKUMA PARÇASI: Teorik Arka Plan ve Matematik

Maalesef teorik derslere zaman ayrılmıyor. Hâlbuki bazı yabancı üniversitelerdeki bilgisayar mühendisliği benzeri bölümlerde tam 2 yıl boyunca **klavyeye dokundurmuyorlar** bile... Öğrenciye teorik arkaplan kazandırıyorlar, onu yetiştiriyorlar. Programlama üzerine ise daha sonra eğiliyorlar.

Bizde ise öğrenci 1. sınıfa başlar başlamaz **programlama ödevleri** veriliyor. Bizim programlamaya ağırlık vermemizin avantajı **hızlı kod yazabilme** ve **hızlı iş bulabilme** olarak ortaya çıkıyor. Ama dezavantajı da var; **teoriklere önem verilmiyor**, orası zayıf kalıyor. Yöneylem, Sayısal Analiz gibi derslerin kalkmış olması da sizin, matematiksel olarak zayıf kalmanıza sebep oluyor.

Oysa çok iyi **matematiksel arka plan** gerekiyor. Discrete, istatistik, stokastik süreçler... Doktora yapıp hoca olduktan sonra bile satır satır Discrete Mathematics dersine çalışıyorlar.

Richard Jhonsonbaugh – Discrete Mathematics: Alın okuyun. Okumazsanız **derslerden geçersiniz, ama bu işi öğrenemezsiniz. Programlama metotlarını** okuyun, nerelerde kullanıldığını öğrenin. Örneğin “Vezir yerleştirme problemi **backtracking yöntemi**yle çözülüyor.” gibi... Yazı değerlendirin, çalışın.

Bölüm kısıtlıdır, her şeyi veremez! Öğrenci daha 2. sınıftayken bunu kendisi fark edip matematik bölümünden, istatistik bölümünden, hatta ODTÜ’den Bilkent’ten ders almalı... Eskiden öyle asistanlar vardı; saatine bakıyor, “Bilkent’te dersim var, 10 dakika var.” deyip, ormandan koşu koşu derse gidiyordu.

Mustafa Ege

AĞAÇ YAPISININ GENELLEŞTİRİLMİŞ LİSTE YAPISIYLA KURULMASI

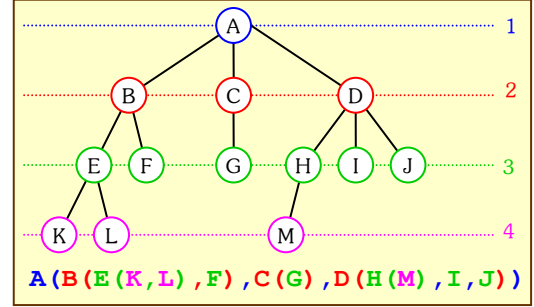
Ağacı **yorum farkıyla** genelleştirilmiş liste yapısıyla kuracağız. Şöyle bir notasyon kullanabiliriz;

C (G) C, G'nin atasıdır.

A (B . . . , C . . . , D . . .) A; B, C, D'nin atasıdır.

A-B-E
| |
| F
|
C-G
|
D-H-M
|
I
|
J

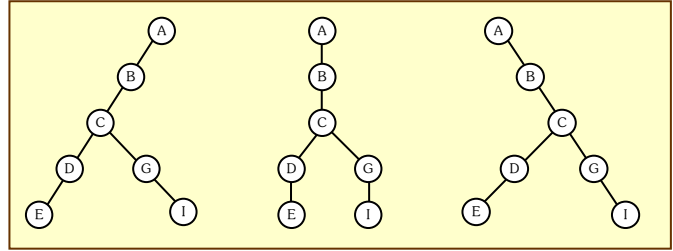
“Ben virgülden hiç haz etmiyorum, nokta kullansam?” Olur. “Benim bu değil de hani o kaşlı parantez var ya, o hoşuma gidiyor.” Olur. **Esnekliği görün, notasyonu size kalmış.** Değişik bir notasyonla da lineer formu elde edebilirsiniz. Bundan ona, ondan buna geçebiliyorsan tamamdır; senin içerdeki yapını kimse bilemez. Oturup o yapının ne kadar güçlü olduğunu birisiyle tartışmadıysan, senin 3 gün, 1 hafta harcadığın o yapını kimse bilemez.



Bu kadar veriyi tutmak için **sadece 1 satırlık text yeterlidir**. Kullanıcının *insert* & *delete* hakkı varsa, oynuyorsa bunu saklamalıyım. Bu tür projeler aldığınızda avantajını belki göreceksiniz. 342’de (Yazılım Mühendisliği Lab.) sorduk, anlayan anladı. Ama ödev olsun diye yazan, *brute brute force* ile yazdı. 8 Puzzle probleminde koca görüntü 1 byte’ta tutulabiliyor. Buna karşılık gelen bir şifreleme yapıyoruz. Sonra deşifre ediyoruz. Çünkü bilgisayar bilimlerinde depolama sorunu önemlidir.

Ağacın Genelleştirilmiş Bir Liste ile Gerçekleştirilmesinin Dezavantajı

Şekildeki üç ağacın da lineer formları aynıdır. Sağ-sol ayrımı yoktur. Ama sol - sağ ilişkisinin verilememesine çözüm olarak düğümlerin yanında L-R diye ek bir bilgi tutulabilir, bu sorun bu şekilde aşılabılır.



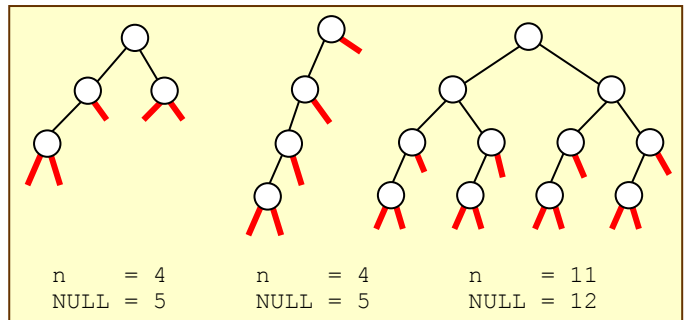
Ödev 1: Verilen listenin bellek görünümünü ve ağaç yapısını çiziniz: **A (B (E (H (I (J, K), F), C (G), D)**

Ödev 2: Şekildeki gibi ağaç yapıları verildiğinde liste yazımını çıktı veren yordamı yazınız.

KILAVUZLU İKİLİ AĞAÇ (THREADED BINARY TREE)

“A year spent in artificial intelligence is enough to make one believe in God.” [Alan Jay Perlis]

n elemanlı bir ikili ağaçta bulunan göstergelerin çoğu **NULL** durumunda bulunur. **İSPATI:** Çünkü her düğümün sağ ve sol çocuk olmak üzere 2 gösterge alanı vardır, yani ağaçta toplam **2n** gösterge bulunur. Kök dışındaki her düğümün babasından kendisine bir gösterge vardır, yani **n-1** tane gösterge doludur. Bu durumda geri kalan **n+1** adet gösterge boştur (NULL değerini içermektedir). [wiki]



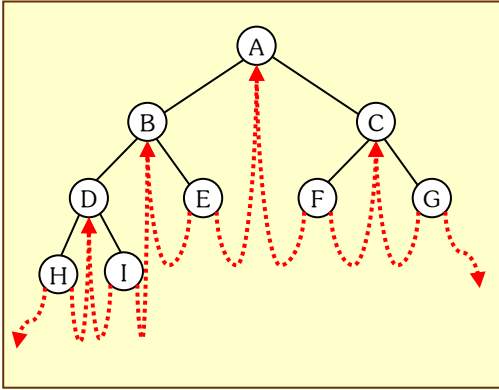
Alan Jay Perlis [w](1922–1990, Carnegie Mellon University & Yale University) ve **Chris Thornton** (University of Sussex) boş göstergelerin ağaçtaki diğer düğümler için kullanılabileceğini göstermişlerdir. Bu göstergelere **kılavuzlu gösterge** adını vermişlerdir. İsimle ararsanız makalesine ulaşmanız mümkündür.

Kılavuzlu İkili Ağaç Kurma Kuralları

p göstergesi ikili ağaçta bir ögeyi gösterebilir.

1) Eğer $p \rightarrow \text{solcocuk} == \text{NULL}$ ise $p \rightarrow \text{solcocuk}$ göstergesi, **kök-ortada dolaşımında p'den önce** ziyaret edilmesi gereken ögeyi gösterecek biçimde kurulur. Diğer bir deyişle, NULL değerini p'nin **kök-ortada önceli** ile (*inorder predecessor*) değiştirmiş oluruz.

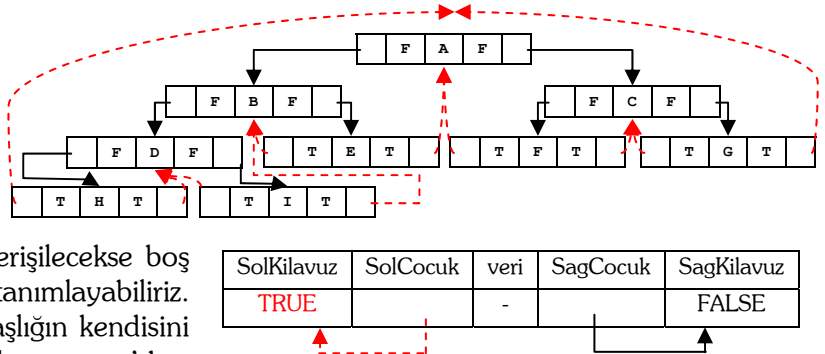
2) Eğer $p \rightarrow \text{sagcocuk} == \text{NULL}$ ise $p \rightarrow \text{sagcocuk}$ göstergesi, **kök-ortada dolaşımında p'den sonra** ziyaret edilmesi gereken ögeyi gösterecek biçimde kurulur. Diğer bir deyişle, NULL değerini p'nin **kök-ortada ardılı** ile (*inorder successor*) değiştirmiş oluruz.



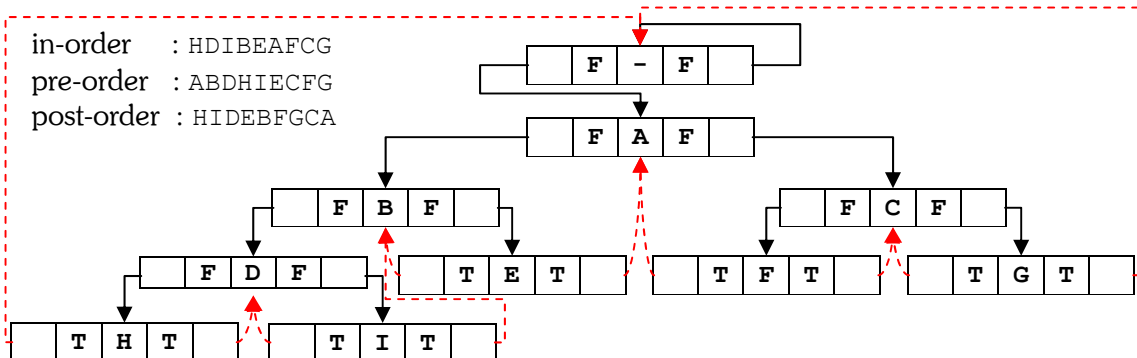
Kılavuzları, normal göstergelerden ayırt edebilmek için **kırmızı renkli noktalı ok** kullandık. Siz hep siyah *pointer* kullanmıştınız. Derleyicide kırmızı *pointer* var mı? Onların kırmızı olduğunu nerden anlayacağız? Fazladan iki bilgi tutmak gerekir. Eğer $p \rightarrow \text{solKilavuz}$ bilgisi TRUE ise $p \rightarrow \text{solcocuk}$ göstergesi bir kılavuzdur, yoksa $p \rightarrow \text{solcocuk}$ sol çocuğu gösterir. Benzer şekilde $p \rightarrow \text{sagKilavuz}$ bilgisi TRUE ise $p \rightarrow \text{sagcocuk}$ göstergesi bir kılavuzdur. Peki, fazladan bilgi tuttuk, bellek karmaşıklığı artmaz mı? Artsın, zaman karmaşıklığı bellek karmaşıklığından daha önemlidir. Kaldı ki, bunu 1'erden 2 bitle bile halledebilirim, çok cüzi bir bellek kullanımı olur. Ama biz kolay olsun diye `short int` tanımlayacağız.

```
1: typedef struct kilavuz_agac *kilavuz_gost;
2: typedef struct kilavuz_agac {
3:     short int solKilavuz; // TRUE ise solCocuk bir kılavuzdur
4:     kilavuz_gost solCocuk;
5:     char veri;
6:     kilavuz_gost sagCocuk;
7:     short int sagKilavuz; // TRUE ise sagCocuk bir kılavuzdur
8: }
```

Buraya dek söylenenlere göre şekildeki ağaç oluştu, ama **iki kılavuz havada asılı kaldı**. Şimdi, bir başlık ögesi sayesinde asılı kalan göstergelerden de kurtulacağız. Eğer L, kılavuzlu ikili ağaçta başlık düğümünü gösteriyorsa ve $L \rightarrow \text{solcocuk}$ ile gerçek ikili ağacın ilk ögesine (A'ya) erişilecekse boş kılavuzlu ağacı yandaki tablodaki gibi tanımlayabiliriz. Ağaç boşken sağ ve sol göstergeler başlığın kendisini gösterir. Sağ kılavuz FALSE'tur. Sol kılavuz TRUE'dur, öge eklenince FALSE yapılacaktır.



Dikkat! İkili ağacı *inorder* dolaşıma göre oluşturuyoruz. *Preorder* dolaşıma göre zaten kuramazsın.



Kılavuzlu İkili Ağaçta Kök-Ortada Dolaşım

p, ikili ağaçta bir ögeyi gösterebilir. Ağacı kök-ortada dolaşabilmek için her bir ögenin kök-ortada ardılını bilmek gerekir. Ardıl, şu şekilde bulunur:

i) Eğer `p->sagKilavuz == TRUE` ise `p->sagCocuk` (kılavuz olduğu için) zaten p'nin ardılıdır.

ii) Eğer `p->sagKilavuz == FALSE` ise `p->sagCocuk`'tan başlanıp `solCocuk` bağları işlenerek `solKilavuz` bilgisi `TRUE` oluncaya kadar ilerlenir. Yani solu olmayan bir düğüme varana dek sağ çocuğun sol bağından itibaren sola gidilir.

SORU: Kök-ortada dolaşıma göre L ile işaret edilen ögenin ardılını bulan `kilavuz_gost_kokortada_ardil_bul(kilavuz_gost L)` yordamını yazınız.

```
1: kilavuz_gost kokortada_ardil_bul( kilavuz_gost L ){
2:     // L ile işaret edilen (NULL olmayan) ögenin ardılını bulur
3:     kilavuz_gost p = L->sagCocuk;; // Ardılını işaret edecek olan gösterge
4:     if( !L->sagKilavuz ) // Sağ Kılavuz FALSE mu
5:         while( !p->solKilavuz ) // Sol Kılavuz TRUE oluncaya dek
6:             p = p->solCocuk; // Sol Çocuk sağından ilerle
7:     return p;
8: }
```

Kodu test edelim: -HDIBEAFCG-

```
1: void kilavuz_kokortada_dolas( kilavuz_gost L ){
2:     kilavuz_gost kokortada_ardil_bul( kilavuz_gost L ); // prototip
3:     kilavuz_gost p = L;
4:     for(;;){
5:         p = kokortada_ardil_bul(p);
6:         if( p == L ) break; // ağaç dolaşılıp başlığa dönüldü mü
7:         printf("%c ", p->veri);
8:     }
9: }
```

Bu iki zât diyor ki; “Kök-ortada dolaşıma göre kurduğum bu ağacı döngüsel olarak hem *postorder* hem de *preorder* dolaşabilirim.” *Preorder*'ı siz yazabilirsiniz, *postorder*'ı yazmak önceki sorduğumdan zordur.

Bir problemin çözümüne yönelik makaleleri okudun, $O(n^2)$ 'de çözülebilmiş. Oturdun, çalıştın ve $O(n^{1.89})$ 'a düşüren bir algoritma yazdın. Büyük iş yaptın! Literatüre senin algoritman geçer.

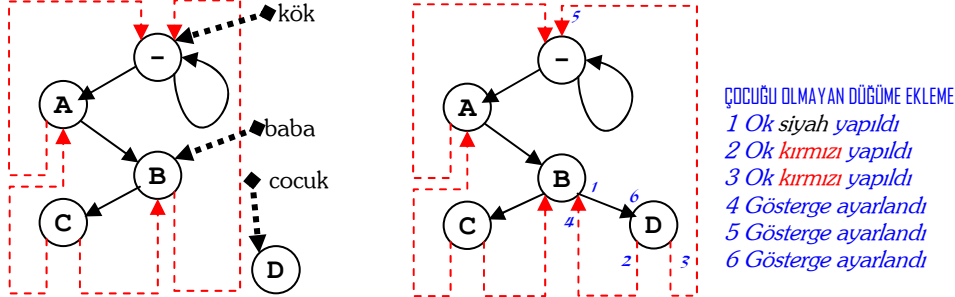
İPUCU:

Kod parçalarının başındaki **satır numaralarını hızlıca silmek için** MinGW'de ya da Microsoft Office'de ALT tuşuna basılı tutarak silmek istediğiniz alanı seçebilir ve DEL tuşuyla silebilirsiniz. Böylece farklı satırlarda olmasına bakılmaksızın sadece seçtiğiniz alan silinecek, satır düzeni bozulmayacaktır.

Kılavuzlu İkili Ağaca Düğüm Ekleme

Ekleme yordamı tek başına kolayca ekleme yapabilecek bir yordam değildir, biraz giriftir. Sağa ve sola ekleme biçimi farklı olduğundan dolayı iki ayrı yordam yazılmalıdır. Şimdi sağa ekleme yordamını nasıl yazabileceğimizi düşünelim.

- 1- Baba ile işaret edilen ögenin **sağ alt ağacı olmasın**. Çocuk ile işaret edilen ögeyi baba ile işaret edilen ögenin sağ çocuğu olarak kılavuzlu ikili ağaca nasıl ekleriz?

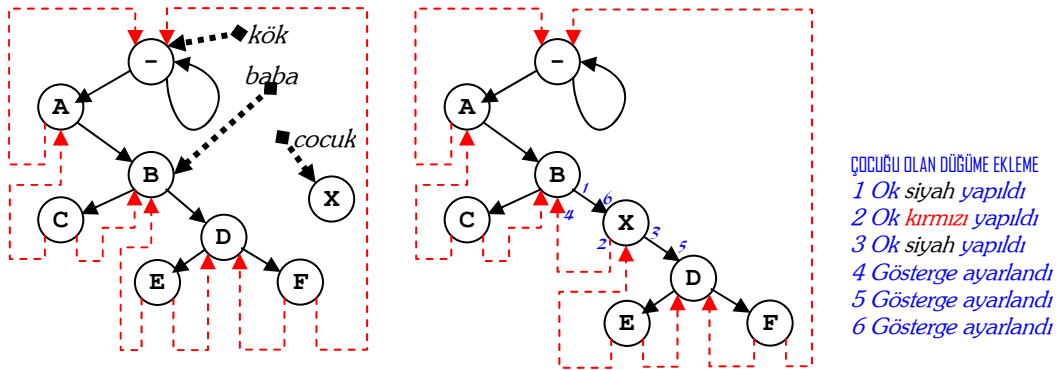


```
1: cocuk->sagCocuk = baba->sagCocuk;
2: cocuk->sagKilavuz = TRUE; // babanın alt ağacı yoktu, çocuğun da olamaz
3: cocuk->solCocuk = baba;
4: cocuk->solKilavuz = TRUE;
5: baba ->sagCocuk = cocuk;
6: baba ->sagKilavuz = FALSE; // TRUE idi
```

6 aktarma komutuyla kılavuzlu bir biçimde ekleme yapabiliriz. 1. aktarma komutunda babanın sağ çocuğu ile işaret edilenin *header* olup olmadığını bilmiyoruz, bizi zaten ilgilendirmiyor.

- 2- Baba ile işaret edilen ögenin **sağ alt ağacı var ise** çocuk ile işaret edilen ögeyi babanın sağ tarafına yerleştirmek, biraz daha dikkatli gösterge değişimini gerektirir.

İkili arama ağaçlarında en az bir çocuğu olmayan düğümlere ekleme yapılabilirdi. Çocuk olmayan noktaya eklediğimizden dolayı bir atama ile bunu başarabiliyorduk. *Peki, burada yapılması istenen şey nedir?* D, B'nin çocuğu iken X'i B'nin çocuğu haline getir; ama D alt ağacı da kaybolmasın, kılavuzlu ağaç yapısı bozulmasın, kırmızı göstergeler yine doğru yerleri göstere. Öyleyse X'in önceli B oldu. D'nin eski önceli olan E, X'in şimdiki ardılı oldu. **Bu durumda çocuk ile işaret edilen öge babanın eski ardılının önceli yapılmalıdır.** Bunu keşsettikten sonra yordamı rahatlıkla yazabiliriz.



```
1: cocuk->sagCocuk = baba->sagCocuk;
2: cocuk->sagKilavuz = FALSE; // babanın alt ağacı şimdi çocuğun alt ağacı
3: cocuk->solCocuk = baba;
4: cocuk->solKilavuz = TRUE;
5: baba ->sagCocuk = cocuk;
6: baba ->sagKilavuz = FALSE; // zaten FALSE idi, bu atamaya gerek yok
```

SONUÇ: Sağa ekleme yaparken sağ çocuğun olup olmamasına göre iki durum oluştu. 2. durum için babanın eski ardılının öncel bilgisini günclemek ve 2 numaralı aktarmayı da uygun şekilde yazmak suretiyle her iki durumun da üstesinden gelebilecek tek bir yordam yazmak mümkündür.

```

1: void saga_ekle( kilavuz_gost baba, kilavuz_gost cocuk ){
2: // çocukla işaret edilen ögeyi babanın sağ çocuk. olacak biçimde kıl. 2li ağaca ekler
3: kilavuz_gost p;
4: cocuk->sagCocuk = baba->SagCocuk;
5: cocuk->sagKilavuz = baba->SagKilavuz;
6: cocuk->solCocuk = baba;
7: cocuk->solKilavuz = TRUE;
8: baba ->sagCocuk = cocuk;
9: baba ->sagKilavuz = FALSE;
10: if( !cocuk->sagKilavuz ){ // baba'yla işaret edilen ögenin alt ağacı var mı
11: p = kokortada_ardil_bul(cocuk);
12: p->solCocuk = cocuk;
13: }}

```

[1:] Bakın, yordamımız açıkça tanımlanmıştır. Adı belli, giden parametreler belli, dönüş değeri belli.

[2:] Açıklama satırlarını unutmuyorsunuz, yeterince açık ve net olmalı. Kodun giriş kısmında baba ve/veya çocuk NULL mı diye test edilmiyor. Bunu durumu da açıklamada belirtebilirsiniz.

NULL mı kontrolü: saga_ekle yordamı, müstakil olarak kullanılmayıp, başka bir yordam içinde kullanılacağı için NULL mı kontrolünü burada yapmaya gerek yoktur. Örneğin, ikili arama ağacına ekleme yaparken sağa mı, yoksa sola mı ekleneceğini bulunca, tam o noktada ekleme yapan aktarmanın yerine saga_ekle ya da sola_ekle yordamını çağıracağız. Bu kontrolü, saga_ekle yordamını kullanan yordamda yapmak gerekir. Ama saga_ekle yordamını müstakil olarak birisi kullanacaksa o zaman bunu da kontrol etmeliyiz.

[5:] Çocuğun sağ çocuk bilgisi babanın sağ çocuk bilgisinden alındı. Böylece NULL da gelse değer korundu. [6:] Çocuğun sağ kılavuz bilgisi babanın sağ kılavuz bilgisinden alındı. Böylece 1. durum için kırmızılık (TRUE), 2. durum için siyahlık (FALSE) korunmuş oldu. [7:] Çocuğun kök-ortada öncülü baba olacağı için sol çocuk göstergesi babaya ayarlandı ve [8:] sol kılavuz bilgisi de TRUE (kırmızı) yapıldı.

[10:] Babanın sağ kılavuz bilgisine FALSE atamak 2. durum için gereksizdi, zaten FALSE idi. Ama ilk durum için gerekiyordu. Bu atamayı her iki durumda da yapmak, kodu yanlış çalışır hale getirmez. Fakat bu satırdan sonra kodun doğru çalıştığına hemen karar vermeyin. Daha işletilecek bir if bloğu var.

[11:] Çocuğun sagKilavuz bilgisi (ki 5. satırdaki atamayla babanın sagKilavuz bilgisinden alınmıştı) FALSE ise if bloğuna giriliyor. Yani babanın sağ çocuğu varsa (2. durum) yapılması gereken işlem yapılacak. Babanın sol çocuğu yoksa (1. durum) sadece fazladan bu test yapılır ve yordamdan dönülür.

Açıklama satırı: Satırın açıklamasında yapılmak isteneni yazdık. Buraya “*babanın sağ çocuğu FALSE mu*” diye yazmayın sakın! Senden sonra gelen adam; “*Ulan, ben onu zaten biliyorum.*” demesin. cocuk->sagKilavuz belli, ünlemi de biliyordur her halde... Öyle açıklama yazmalısın ki, gerektiğinde “*Yapmak istediği doğru, ama kodu yanlış yazmış.*” diyebilisin. Açıklama satırları, kodda söylenenden farklı şeyler de söylesin.

Tek yordama indirmek: Dersimizin başından beri söylemeye çalışıyorum; fonksiyon, açıkça tanımlanmış olmalıdır. Öncelikle 3–5 farklı durum olduğunu analiz ettikten sonra yordam isimleri, dönüş değerleri, parametreleri vs. belirlenecek. Her duruma ayrı ayrı yordam yazsak, işin tadı kaçacak; cocukYok_sagaEkle, cocukVar_sagaEkle, cocukYok_solaEkle, cocukVar_solaEkle. Karmaşıklık artırmadan bunları tek yordama indirebilirsek, iyi! Burada yaptığımız gibi yapılacak işleri ortak şekilde tanımlayarak birleştirmeliyiz. Yoksa “*İç içe if... if... deriz, tüm durumları tek yordamda yazabiliriz.*” Ondan kaçın. Öyle olacağına ayrı ayrı yordamlar olsun.

Yazılım Mühendisliği Kavramları: Bakın, kod bölük pörçük değil; cocukVar_sagaEkle, cocukYok_sagaEkle diye **dağılmadık; sade ve okunabilir, bakımı kolay.** Bunlar yazılım mühendisliği kavramlarıdır. Yazılım mühendisliği kavramlarını keşke 1. sınıfta öğretseler, ama çok erken olur. Programlama bilmeyen adama kodun sadeliğinden, okunabilirliğinden bahsetmek... Bu bölüm bana göre 6 yıl olmalı.

SORU: Siz de sola ekleme yordamını düşününüz.

Kod yazarken teyakkuz! Her an dinamite basabilirsin. “*Bunu yapıyorum, ama doğru mu yapıyorum?*”

YIĞINLAR (HEAPS)

Yığınlar, öncelik kuyruğunu oluşturmada kullanılan veri yapılarından biridir. Evvelâ, “*Öncelik kuyruğu ne tür bir kuyruktur?*” sorusunu ele almak gerekir. Bu tür kuyruk veri yapısında kuyruktan ilk alınacak (silinecek) eleman en yüksek ya da en düşük değere sahip değerdir.

Eğer en yüksek değer öncelikli ele alınması isteniyorsa *max priority queue*, en düşük değer öncelikli ele alınması isteniyorsa *min priority queue* söz konusudur. *Priority queue* diyoruz, bakalım kuyruğu ne kadar biliyoruz? Bazen bir şeyleri bilmek de sakıncalı olabilir. Kuyruk diye bir şey biliyorsun, bunda da kuyruk kelimesi geçiyor. Ama bildiğimiz kuyrukta sondan eklenir, baştan alınır. Bu ise ondan farklı...

Öncelik kuyruğuna neden gereksinim duyarız?

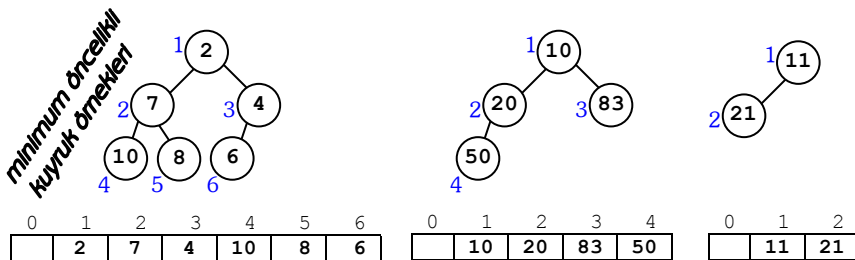
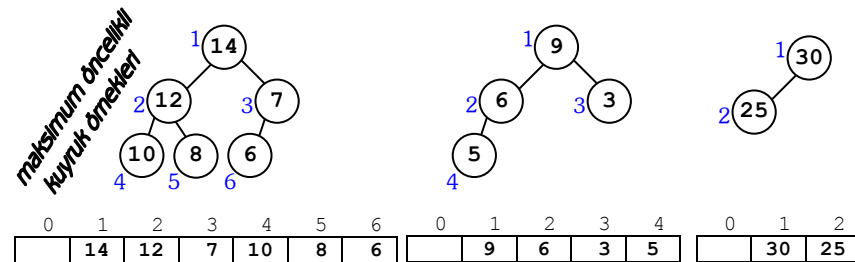
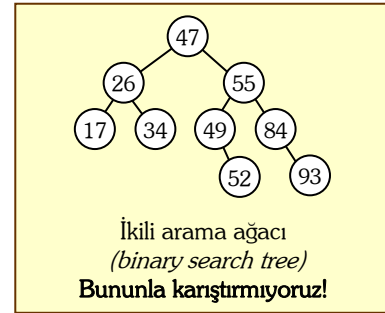
- 1) Bir makinenin (örneğin fotokopi makinesi) servisini sattığımızı farz edelim. Makineyi kullanmak isteyen kullanıcılar (3 sayfa da çekse, 30 sayfa da çekse aynı) sabit bir ücret ödedikten sonra makineyi kullanma kuyruğuna girsinler. Hâliyle her kullanıcının makineyi kullanma süresi de farklı olur. Kazancımı maksimum yapmak istiyoruz. Kullanma süresi minimum olana öncelik vererek kazanç maksimize edilir. (Çakaall!) Minimum öncelikli kuyruk veri yapısı kullanılmalıdır.
- 2) Bir havaalanına inmek isteyen uçakların inişini denetlemek istiyoruz. İniş talebinde bulunan birden çok uçak varsa yakıtı az olana öncelik verilmelidir. “*Kuyruğun arkasındasın. Yakıtın az, ama kuyruk böyle. Ne yapalım?*” diyemeyiz. Minimum öncelikli kuyruk veri yapısı kullanılmalıdır.

Kolay anlaşılabilirliği sebebiyle bu örneklerle anlatıyoruz. Yığınlar, eğer nerede ve nasıl kullanılacağını kavrayabilirseniz, çok çok önemli bir veri yapısıdır. Örneğin işletim sistemi, işlemleri öncelik kuyruğuna koymaktadır.

Öncelik kuyruğu oluşturmada farklı veri yapıları benimsenebilir. Yığınlar bunlardan sadece biridir. **Tam ikili ağaç**, yığıt kurmak amacıyla kullanılabilir. Yığınlar ise bir dizide gerçekleştirilebilir.

Yığını dizide tam ikili ağaç yapısını kullanarak gerçekleştirdiğimizde yığın, dizinin 1. elemanından başlar, 0. indis kullanılmaz. Dizi sıralı değildir, yalnız 1. indisteki elemanın en öncelikli (ilk alınacak) eleman olduğu garanti edilir.

Yığın denince aklımıza *complete binary tree* gelecek, *search tree* değil. Hatırlayalım; **tam ikili ağacın** tüm düzeyleri dolu, son düzeyi ise soldan sağa doğru doludur. **İkili arama ağacı** ile yığın arasında ise bir bağlantı yoktur. “*Hocam bir ikili arama ağacı çizer misiniz?*” Genel istek üzerine çizelim, **bununla karıştırmıyoruz! → şekle bakın...**

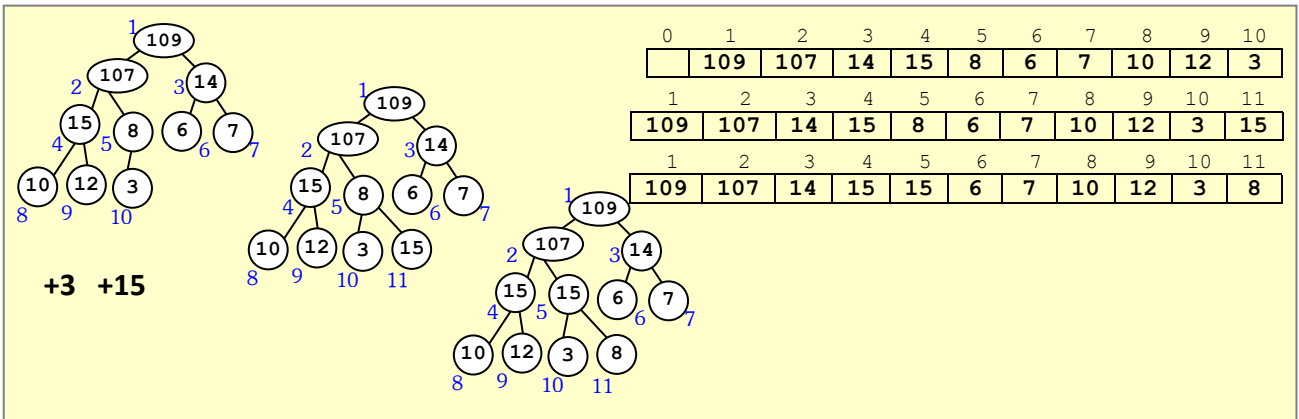
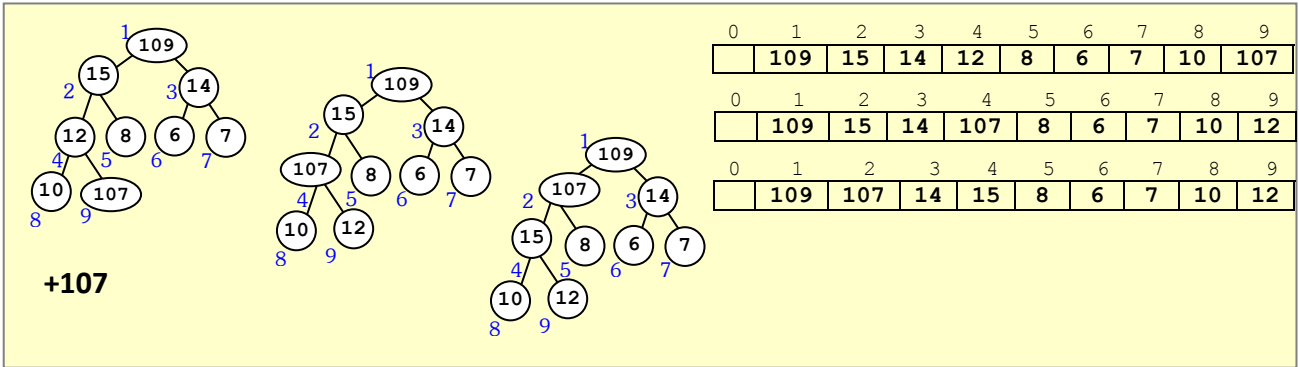
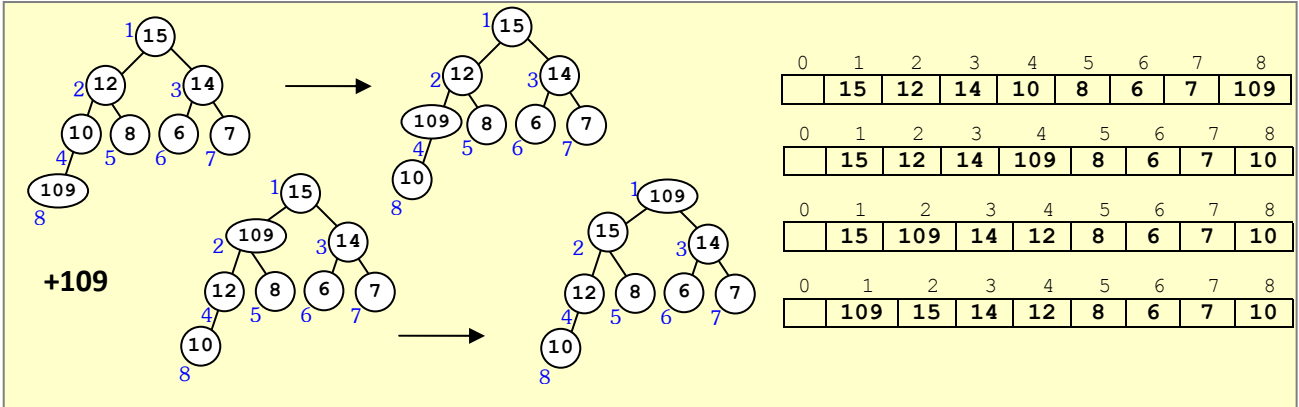
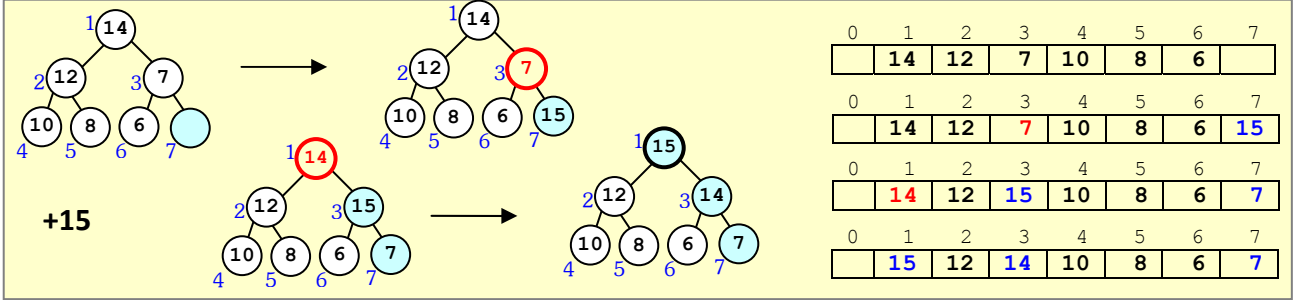


Tanım: Tam ikili ağaçtaki her düğümün değeri çocuklarından küçük değilse söz konusu ikili ağaç **maksimum yığın** (*max heap*) olarak isimlendirilir. (*Maksimum öncelikli kuyruk* kullanabilmek için bu tanım verildi.)

Tanım: Tam ikili ağaçtaki her düğümün değeri, çocuklarından büyük değilse söz konusu ikili ağaç **minimum yığın** (*min heap*) olarak isimlendirilir. (*Minimum öncelikli kuyruk* kullanabilmek için bu tanım verildi.)

Maksimum Öncelikli Kuyruğa Öğe Ekleme

Eklenecek elemanın yeri kesinlikle hazır! Bundan kuşku yok. **Önce o yere eklersin. Sonra** eyvahl! Çocuğundan büyük olmak zorundaydı. Öyleyse **uygun konuma kaydırırız**. Örnek eklemeleri inceleyiniz. Sırasıyla **15, 109, 107, 3, 15** değerleri eklenecektir. En iyi durumda $\Omega(1)$ 'de de ekleme yapılabileceğini görüyoruz (3'ün eklemesi). Mevcut tüm elemanlardan daha büyük bir öğe eklendiğinde ise yeni öğenin köke kadar kaydırılması gerekeceği için $O(\lg n)$ 'de ekleme yapıldığını görüyoruz (109'un eklenmesi). Yani karmaşıklığın üst sınırı $\lg n$, alt sınırı ise 1 olur. Gerçekte karmaşıklık bu ikisi arasında değişebilir. **Bu durumda zaman karmaşıklığı $O(\lg n)$ 'dir.**



Maksimum Öncelikli Kuyruğa Öğe Ekleme Yordamı

```
1: #define eleman_sayisi 200 // maksimum eleman + 1
2: #define YIGIN_DOLUMU(n) ( n==eleman_sayisi - 1 )
3: #define YIGIN_BOSMU(n) ( !n )
4:
5: typedef struct {
6:     int x;
7:     // diğer alanlar
8: } eleman;
9: eleman yigin[eleman_sayisi - 1];
10: int n = 0;
11:
12: void maksimum_yigin_ekle( eleman item ){
13:     if( YIGIN_DOLUMU(n) ){
14:         Yigin_Dolu(); // Hata iletisi
15:         return;
16:     }
17:     int i = ++n;
18:     while( i != 1 ){ // i==1 ise kökteyiz
19:         // eklenecek değer atasından küçükse yerleşim yeri bulundu, döngüden çık
20:         if( item.x <= yigin[i/2].x ) break;
21:         yigin[i] = yigin[i/2]; // atayı i konumuna kaydır
22:         i /= 2; // bir üstteki ataya bakmaya hazırlan
23:     }
24:     yigin[i] = item;
25: }
```

Maksimum Öncelikli Kuyruktan Öğe Alma/Silme Yordamı

Sadece bir öğeyi almak kolay! Ama iş burada bitmiyor. Öğeyi silmeli ve *max* öncelikli kuyruk yapısını da korumalıyız.

```
1: eleman maksimum_yigin_al( void ){
2:     if( YIGIN_BOSMU(n) ){
3:         Yigin_Bos(); // Hata iletisi
4:         return 0; // Burada derleme hatası var. eleman tipinde bir değer dönmeliydi
5:     }
6:     int i, j;
7:     eleman x = yigin[1]; // x, döndürülecek değerdir
8:     eleman deger = yigin[n--]; // deger için ağaçta uygun yer aranacak
9:     for( i=1, j=2 ; j<=n ; ){
10:         if( j<n ) // sonraki satırda [j+1] indis taşmasına önlem
11:             if( yigin[j].x < yigin[j+1].x )
12:                 j++; // j'de büyük değerli çocuk var
13:         if( deger.x >= yigin[j].x ) // büyük çocuktan daha büyük mü
14:             break; // kaydırma yapmadan çık
15:         yigin[i] = yigin[j]; // değer'le karşılaştırıldı, taşıma yapılabilir
16:         i = j;
17:         j *= 2;
18:     }
19:     yigin[i] = deger;
20:     return x;
21: }
```

[5:] Yordamın görevini yapmadığını belirtmek üzere **sıfır döndürmek**, [20:] yaptığında ise ilgili elemanı döndürmek yerine yordamı daha güzel yazmak mümkündür. Yordam *boolean* tipte yazılarak görevini başarılı yaptığında *TRUE* dönmesi, başarısızlık durumunda ise *FALSE* dönmesi sağlanabilirdi. Referansla aktarılan bir parametre ile de döndürülmesi istenen eleman döndürülürdü. *n* değerini de referansla aktarmak anlamlı, hoş olurdu.

Fundamentals of Data Structures – Ellis Horowitz: Kitaptaki kodlar zaten çok doğru kodlar değil. Ama tanımlar doğru olduğundan, benim gördüğüm kadarıyla veri yapılarını öğrenmek için son derece ideal bir kitap...

Maksimum Öncelikli Kuyruktan Öğe Alma/Silme

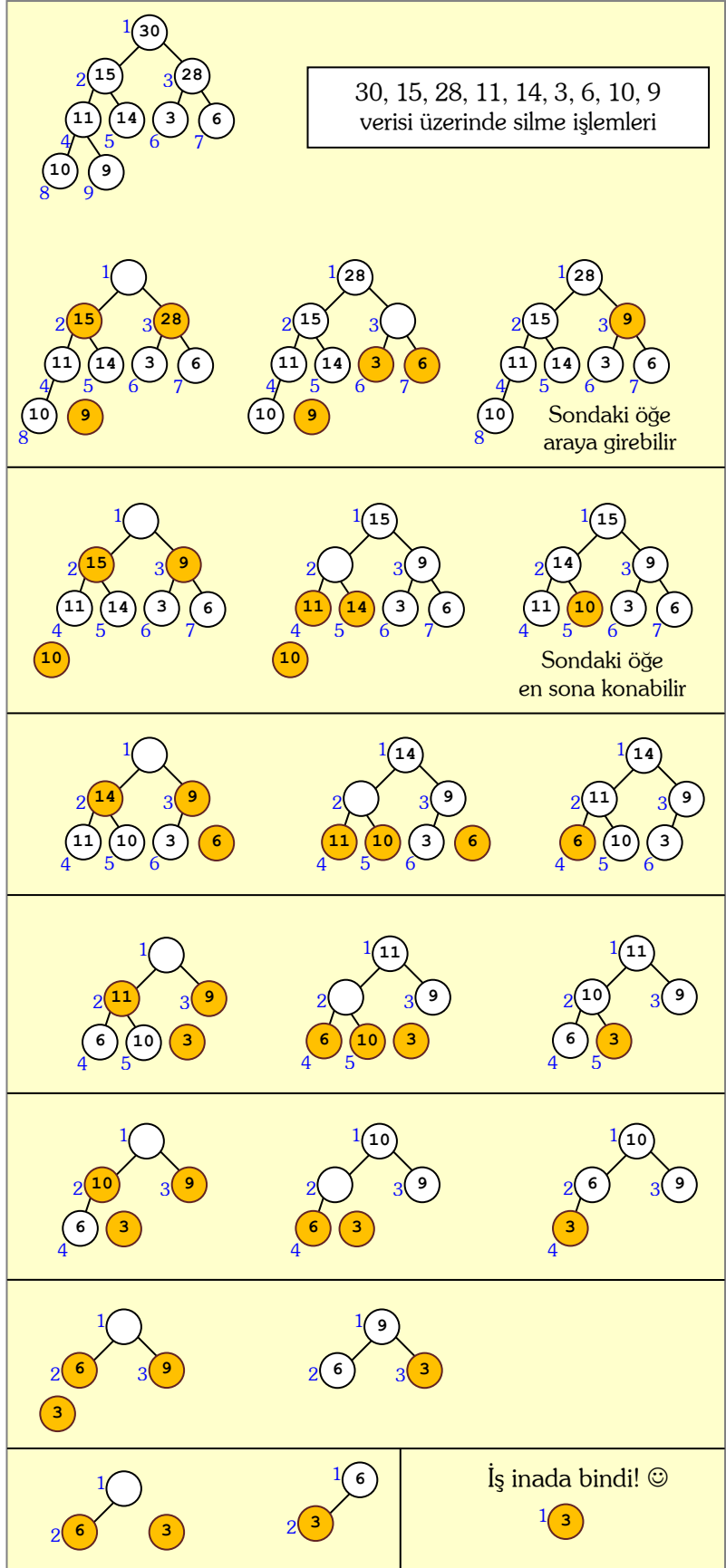
Yığın dizisiyle gerçekleştirmiştik. Yığının tanımı gereği **ilk alınacak eleman, dizideki ilk elemandır**. Bunun yanında “tam ağaç” yapısını bozmamak için en **sondaki elemanı** da uygun bir konuma **kaydırmalıyız**. Bu kaydirmayı yaparken maksimum yığın yapısını korumaya dikkat etmeliyiz.

Her adımda (iki çocuk ve sondaki eleman olmak üzere) dikkat edilecek 3 değer var. İki gösterge gibi; i ile, en sondaki elemanı kaydıracağım konumu ararken; j ile de üzerinde bulunduğum düğümün çocuklarını kontrol ediyorum. Bu üç değerden (iki çocuk ve sondaki eleman) en büyük olanı, i 'nin gösterdiği yere taşıyorum. Sonra da taşınan değer eski konumuna ilerleyip aynı kıyaslamaları yapıyorum. Tâ ki, sondaki düğüm bir yere yerleşene kadar.

En sondaki öğe bazı durumlarda arada bir yere girebilir; en kötü durumda ise arada bir yere yerleşemez, en sona konması gerekir. $\rightarrow O(\lg n)$.

Min halleri: *Max* için yazdığımız bu yordamları *min* için yazılmış hale çevirmek çok kolaydır, ona rahatlıkla geçebiliriz. Adını değiştirip küçük-büyük işaretlerini değiştirmek yeterlidir. Yordamın adını değiştirip küçük ve büyük işaretlerini de büyük tersine çevirirseniz **başarılı bir şekilde min heap yazmış olursunuz**. Bu yordamların hepsini dinamik eleman olarak yazabiliriz. Ama kitapta da çalışıyorsanız paralel gitmek için...

Problemi iyi yakalamak: Her uçağın saniyede harcayacağı yakıt farklıdır. Her saniyede kesme (*interrupt*) oluşturup yakıtları güncle. Gerçek durumda her saniyede günclenmeli. Tehlikeli durumda kırmızı ile falan gösterilmeli. Bir uçağın bir saniyede 10 galon (37,86 litre) yakıtı giderken diğeri 10 galonla bir saat uçabilir belki.



Yığın Diğer Veri Yapılarıyla Karşılaştırılması

Niçin yığın kullanıyoruz? Basit bir dizi ya da bağlılı liste kullanılamaz mı? (*Hatırlatma*: en iyi (Ω) ve en kötü (O) durum aynı derecedense bunu ortalama durum (Θ) ile ifade ediyorduk.)

- 1) **Sırasız dizi (*unordered array*)**: Dizi sonuna ekleme bir çırpıda yapılır, $\Theta(1)$; en büyük elemanı bulmak için arama yapmak gerekir, $\Theta(n)$. Sırasız ya, tamamına bakmalısın, başka şansın yok. “En büyük değeri bir *max* değişkeniyle tutsak?” O uçak inince ne yapacaksın? Öyleyse sıralı dizi kullanmalıydın. Elemanı ekleyip sıralarsak bile, en hızlı sıralama algoritması $n \cdot \lg n$ ’dir.
- 2) **Sırasız liste (*unordered list*)**: Ekleme hızlandırmak için listenin sonuna değil de önüne ekleyebiliriz, $\Theta(1)$; en büyük elemanı bulmak için tıpkı sırasız dizideki gibi hepsini dolaşmak gerekir, $\Theta(n)$.
- 3) **Sıralı dizi (*ordered array*)**: Dizide uygun yere ekleme $O(n)$; silmeyi hızlandırmak için (*max heap* için konuşuyoruz) diziyi küçükten büyüğe sıralı tutarız, $\Theta(1)$. Kötü algoritma şöyle yazılır: Sıradan hepsini arar, sıralı olma avantajını kullanmaz. Hâlbuki *binsearch* yapmalıydı. Böylece arama maliyeti $O(n)$ ’den $O(\lg n)$ ’e indi. Bir de n kaydırma yapılacak. Karmaşıklık $\lg n$ ile n arasında değişir. *binsearch* kullanmazsan $O(n)$ değil $\Theta(n)$ olurdu.
- 4) **Sıralı bağlılı liste (*ordered link list*)**: Ekleme bir çırpıda da yapılabilir, n düğümü dolaşmak da gerekebilir, $O(n)$; silmeyi hızlandırmak için (*max heap* için konuşuyoruz) listeyi büyükten küçüğe sıralı tutarız, $\Theta(1)$. Böylece listenin başından hemen silip, liste başını güncleyebiliriz.
- 5) **Maksimum yığın (*max. heap*)**: Hem ekleme hem de silme $O(\lg n)$

5 farklı yaklaşımı hiç kod yazmadan, tuşlara dokunmadan analiz ediyorum. Hocanın anlattığı benim de tam anlamadığım bir aşağı in, bir yukarı çık yapılacak o yapıya ne gerek var? Bir görelim.

432 (Algoritma Analizi) dersini alsaydınız bu maliyet tablosu sizin için yeterli olacaktı. Ama almadığınız için bunu sayısal verilere dökeceğiz. Ben 2 milyon sayısını çok seviyorum, çünkü logaritmasını almak kolay. O yüzden onu örnek veriyorum. Bir komutun bir saniye sürdüğünü farz edelim. Hadi kümülâtif toplam yapmıyorum da, *max heap* yapısıyla gerçekleştirdiğim programı **şaka olsun diye 100.000 kez çalıştırıyorum. Senin bunu 1 kez çalıştırman kadar sürüyor.**

	MALİYET		ZAMAN	
	EKLE	SİL	EKLE	SİL
1	$\Theta(1)$	$\Theta(n)$	1	2.000.000
2	$\Theta(1)$	$\Theta(n)$	1	2.000.000
3	$O(n)$	$\Theta(1)$	2.000.000	1
4	$O(n)$	$\Theta(1)$	2.000.000	1
5	$O(\lg n)$	$O(\lg n)$	20	20

“Komut saniye değil, mikro saniye düzeyindedir.” diyebilirsiniz. Bunlar belki *real-time* uygulamalarda kullanılmayacak, ama bir işletim sisteminin parçası olduğunda aralarında ne kadar fark var! “*Kardeşim! Havaalanına yarım saatte, bir saatte bir uçak geliyor. Kim anlayacak benim algoritmanın kötü olduğunu?*” Evet, bir yerde çalışabilirsin, iyi de para alıyor olabilirsin. Ne yazan bunların farkındadır, ne de işveren... Uzmanlık isteyen işlerde ise o şekilde yazılmış kodun akademik ve bilimsel hiçbir değeri yoktur.

Problem böyle olmalı: *Bir eleman ekle, en küçüğünü hemen ver.* Bağlılı kullanıp yok anasına bak, yok babasına bak... Hocam bana sıralı bağlılı liste hoş geldi diyen varsa hala, yapacak bir şey yok.

İleri Düzey: Bu yapı *binary heap*. Bundan daha ileri düzeyde *fibonacci heap*, *binomial heap* algoritmaları var. Daha hızlanıyor. Adamlar $\lg n$ yeter dememişler.

Citius! Altius! Fortius!

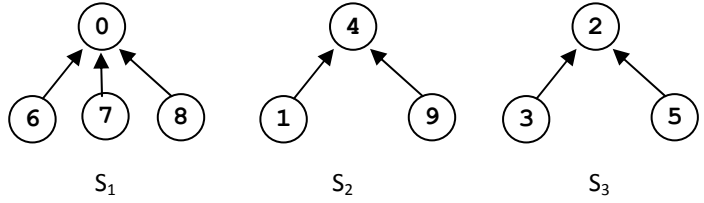
(Latince: Daha hızlı! Daha yükseğe! Daha güçlü!)

[Olimpiyat Oyunları'nın resmi sloganı]

KÜMELERDE “UNION” ve “FIND” İŞLEMLERİ

⇒ Çok çok temel bir veri yapısıdır, ama birçok problemde kullanılır. Öğrenmelisin ki, kullanabilesin. [\[wiki\]](#)

Ağaçları kullanarak kümelerin bilgisayar ortamında nasıl gerçekleştirilebileceği konusyla ilgilenelim. Basitleştirerek $\{0, 1, \dots, n-1\}$ sayılarıyla uğraşalım. Gerçekleştirilecek kümelerin ikili olarak ayrık (ikili kesişimleri boş küme - *pairwise disjoint*) olduğunu da kabul edelim: $S_1 \cap S_2 = S_2 \cap S_3 = S_1 \cap S_3 = \{\}$.



N=10 için elimizde şu kümeler olsun:

$$S_1 = \{0, 6, 7, 8\}, S_2 = \{1, 4, 9\}, S_3 = \{2, 3, 5\}$$

Olası gerçekleştirmelerin her birinde her bir küme şeklindeki gibi ağaç veri yapısı olarak düşünülebilir. Burada çocuklardan atalara bağ vardır. **Kümeler üzerinde yapılacak işlemler:**

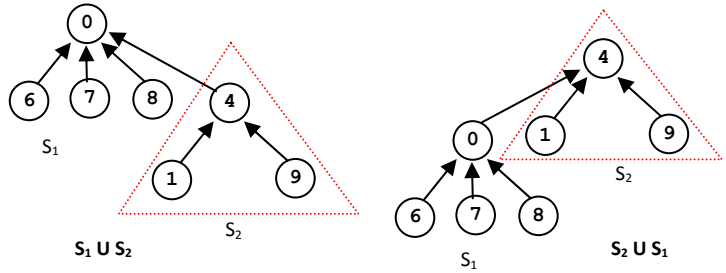
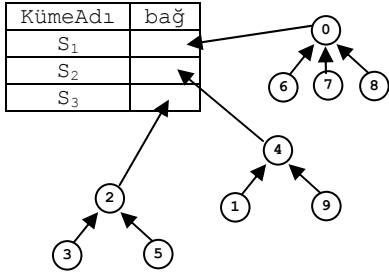
1- **Ayrık küme birleşimi (disjoint set union):** S_i ve S_j iki ayrık küme ise birleşimleri şu biçimde bulunabilir; $S_i \cup S_j = \{x \in S_i \vee x \in S_j\}$. Birleşme sonrasında kümeler bağımsız olarak bulunmayacaktır;

$$S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}.$$

2- **Küme_bul(i):** i elemanını içeren kümeyi bulmak olarak tanımlansın.

Bağlar, çocuklardan atalara doğru olduğundan birleşme iki biçimde olabilir. Köklerden birinin baba bilgisinin diğer kökü işaret edecek şekilde dengelenmesi ile birleşim kolayca gerçekleştirilebilir.

Eğer kümeler bir ad (isim) içeriyor ise gerçekleştirim, aşağıdaki gibi yapılabilir. Yani, amacımız “ S_1 ” diye kümenin adını söylemekse gelip metin olarak görebiliriz.



Ama Birleşim ve Küme_bul algoritmalarını incelerken biz, her küme adını ağacın kökü ile ilişkilendireceğiz. Örneğin “ S_1 ” kümesi yerine “0 nolu küme” diyeceğiz.

Gerçekleştirim: Liste ile gerçekleştirirsek, birleşim işlemini $O(1)$ 'de yapabiliriz. Fakat arama işlemi $O(n)$ 'e çıkar. Başka bir teknik? Mesela, arama maliyetini azaltmak için sıralı tutsak? Her birini sıralamak $O(n \cdot \lg n)$ 'dir, o da maliyetli. **En güvenilir, en sağlam, en hızlı veri yapısı olan diziyi kullanarak gerçekleştirmenin bir yolu var mı?**

Ağaçlardaki düğümler 0'dan n-1'e kadar numaralandırıldığından düğüm numaralarını indis olarak kullanabiliriz. Dolayısıyla her düğüm yalnızca bir alana gereksinim duyacaktır. *ata* dizisinin her bir indisi ilgili düğümün atasına bir bağ gibi düşünülmelidir. Örneğin; 1 ve 9'un atası 4; 3 ve 5'inki 2; 6, 7, ve 8'in atası 0; 0, 2 ve 4 ise kümelerin kökleri.

ata	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	-1	4	-1	2	-1	2	0	0	0	4

```
1: int Kume_Bul_1( int i ){
2:     for( ; ata[i] >= 0 ; i = ata[i] );
3:     return i;
4: }
5: void Birlesim_1( int i, int j ){
6:     ata[i] = j; // i kümesini j'ye bağla
7: }
```

i değerinin hangi kümeye ait olduğunu bulmak için `küme_bul(i)` algoritması negatif bir değer buluncaya dek ilerlemelidir. `Birleşim(i, j)` algoritması da i ve j kümeleri yordama parametresi olarak aktarıldıktan sonra her hangi birisi diğerinin alt ağacı olacak şekilde günlenebilir.

Bakın! Önce problemi bir matematikçi gibi, programlama bilmeyen birisi gibi ele alıyoruz; Öncelikle problemi çizge teorisindeki terminoloji ile doğru ifade edebilmeliyiz.

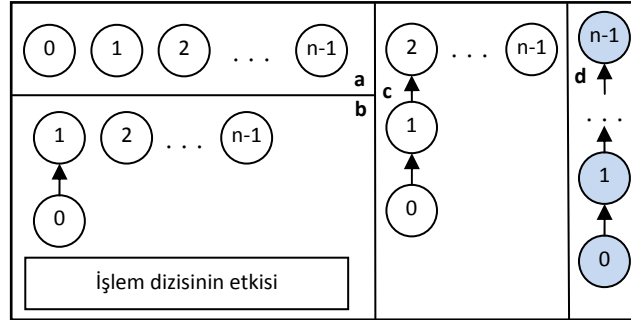
Küme_Bul_1 ve Birlesim_1 Yordamlarının Analizi

$0 \leq i < n$ olmak üzere; n adet kümeyle başlayalım. $S_i = \{i\}$. Başta $ata[i] = -1$ olur. (Tümünde -1 var)

ata	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Sırasıyla işletilecek komutlar:

```
Birlesim_1(0,1);
Birlesim_1(1,2);
Birlesim_1(2,3);
...
Birlesim_1(n-2,n-1);
Küme_Bul_1(0);
Küme_Bul_1(1);
...
Küme_Bul_1(n-1);
```



$$\sum_{i=2}^n i = O(n^2)$$

Bu işlem dizisi, ağaç yapısını bozacaktır. Birlesim_1'in bir kez işletim maliyeti sabittir. Dolayısıyla Birlesim_1'in $n-1$ kez işletimi toplam $O(n)$ kadar süre ister, bunda sorun yok. Bu komutlardan sonra ağaç, şekildeki (en sağdaki d şeklindeki) hale gelmiştir. Artık derinlik n 'dir. Küme_Bul_1 yordamında $(n-i)$. düzeyde bulunan i elemanı için $O(i)$ kadar bir süre gerekir. Yani for döngüsü ağacın ağırlığı kadar çalışır; 0 için $n-1$ kez, 1 için $n-2$ kez... Eğer birleşim böyle olursa, Küme_Bul_1 yordamının karmaşıklığı $O(n)$ 'e gidebilir, kümülatif değerlendirme yaparsak toplamda $O(n^2)$ 'ye çıkar.

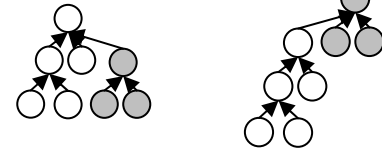
Software Testing: Worst case analizinde kodu, çakılacağı ya da işletiminin en uzun süreceği verilerle test etmeliyiz. Bunu saptamak ise zordur. Beta sürümleri bu yüzden çıkıyor. Beta sürümü ne demektir? “Biz kafasızız, gözden kaçırdığımız hatalar olabilir. Kafasızlığımızı affedin, hata bulursanız bizi bilgilendirin ki, onun üstüne gidelim.” Software Testing, ayrı bir alandır, onun üzerine de makaleler okumalısınız.

Ağırlıklı Kural (Weighting Rule)

Birleşim için ağırlıklı kural (*weighting rule*) uyguladığımızda hem birleşim, hem de küme bulma için daha etkili yordamlar geliştirmiş olacağız.

Tanım: Ağırlıklı kural: i ağacındaki düğümlerin sayısı j ağacındaki düğümlerin sayısından daha az ise j , i 'nin atası; diğer durumda i , j 'nin atası yapılır. (Düğüm sayısı çok olan ata oluyor.)

Küçük, büyüğe eklenirse Büyük, küçüğe eklenirse



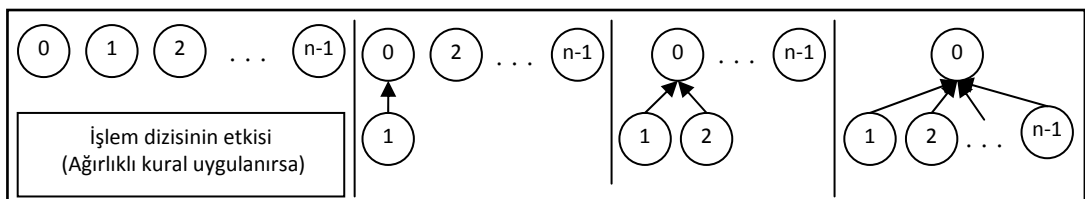
Düğüm sayısı az olanı, çok olana ekliyoruz.
Aksi halde derinlik artabilir.

Sual: Acaba her kümede (ağaçta) kaç düğüm olduğu bilgisi için ayrı bir dizi mi tutmamız gerekir?

El-Cevap: Kökler dışındaki tüm düğümler için ata dizisinde pozitif değerler tutuluyordu. Öyleyse köklere ilişkin düğüm sayısı bilgisi de ata dizisinde negatif olarak tutulabilir. Ekstra diziye gerek yoktur.

```
1: void Birlesim_2( int i, int j ){
2:     // Ağırlıklı kurala göre i ve j kümeleri birleştirilir
3:     int p = ata[i] + ata[j]; // eleman sayısı, kümelerin eleman sayıları toplamı
4:     if( ata[i] > ata[j] ){ // i daha az düğüme sahip (negatif değer olduğundan)
5:         ata[i] = j; // i kümesini j kümesine bağla
6:         ata[j] = p; // j kümesinin eleman sayısını günle
7:     } else { // j daha az düğüme sahip ya da eşit
8:         ata[j] = i; // j kümesini i kümesine bağla
9:         ata[i] = p; // i kümesinin eleman sayısını günle
10:    }
```

Ağacın derinliği n 'den 1'e düştü.



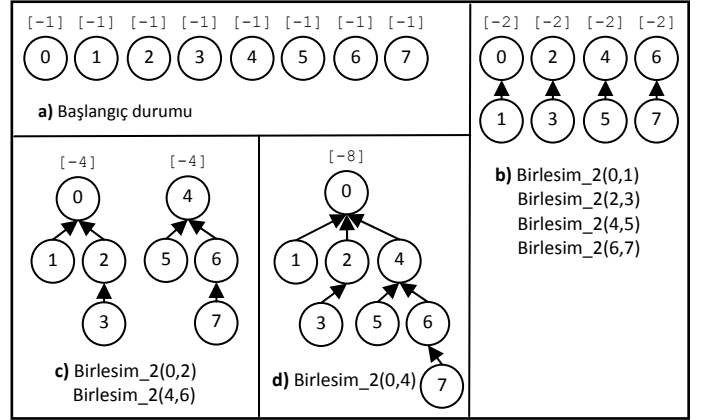
Soru: Ağacı ne kadar dejenere eder, bozar?

Algoritmanızı doğru, etkili ve yazılım mühendisliği kurallarına göre yazacaksınız. Ama bitmiyor, en çok bozacak veriyi sorguluyoruz. Arka planda ne olursa, kodum en fazla çalışır? **Bilgisayar bilimi, bilgisayar mühendisliği işte budur!** Programcıysan eyvallah, sana bir çözümümüz yok zaten. Ama bilgisayar bilimlerini öğrenmek isteyen, bunları anlatmak durumundayız. Örneğin matematik bölümü mezunu, istatistik mezunu ve bilgisayar mühendisliği mezunu aynı işe müracaat edecekler. Soracaklar: “*Hepiniz C, Java vs. biliyor musunuz?*” Evet, biliyorlar. “*Tamam, hepiniz işe girin, aynı maaş!*” Sen diyeceksin ki: “Bir dakika! Ben bilgisayar mühendisiyim. Nasıl aynı maaş olur? **Sen bir soru sor bakalım!** Bir soru sor, nasıl çözdüğüme bak!” Hemen de maaşa bağladık. Böyle bu işler, ticaret tabi...

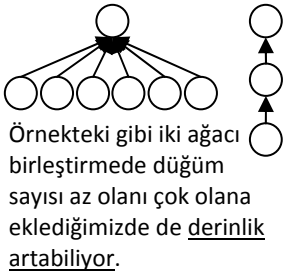
Lemma: T, Birlesim_2 yordamıyla çeşitli kümelerin birleşimi gerçekleştirildikten sonra elde edilmiş olan m düğümlü bir ağaç ise T'nin derinliği $\lfloor \lg m + 1 \rfloor$ değerinden büyük olamaz.

Matematik bölümü olsaydık “İspatlayınız.” diye soru devam edecekti, Tümevarımla (*induction*) ispatlanıyor. Her seferinde veri kümesini ikiye ayırıp, birini diğerine bağlarsak en kötü durumu elde ederiz, her seferinde derinlik 1 artar. Sonuçta derinlik $\lg m$ olur. (Düğüm sayısı 7 ya da 9 gibi 2'nin tam kuvveti olmayan bir sayıysa o zaman $\lg m + 1$ gelebilir. Sonuçta maliyet logaritmiktir; $O(\lg n)$.)

Algoritmayı anlattık, “*Hangi sırada uygulanırsa maliyeti en fazla artar?*”ı da anlattık. Bakın, karmaşıklığı n 'den $\lfloor \lg n \rfloor + 1$ 'e düşürdük. **Algoritma geliştirme bu, işte!** Ama yerimizde durmuyoruz, devam...



Azaltma Kuralı (Collapsing Rule)

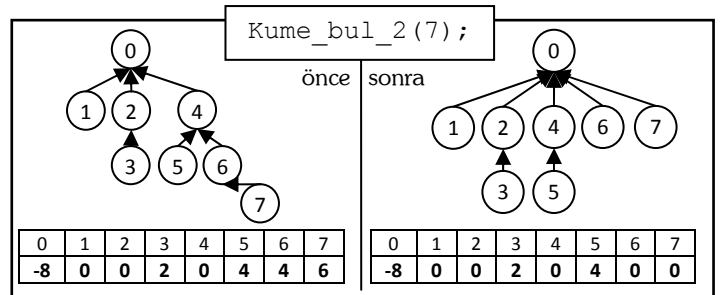


Son durumdaki yapı oluştuktan sonra Küme_bul(7) işletilirse ata dizisinde 3 kez ilerledikten sonra köke ulaşabiliriz. Söz gelimi programın ihtiyacı gereği Küme_bul(7) tekrar tekrar işletilecek olursa toplamdaki maliyet katlanacaktır.

Düğüm sayısı az olanı çok olana eklessek bile, bazen derinlik artabilir. Düğüm sayısı yerine derinliği göz önünde tutmak daha anlamlı olurdu, ama yapımız derinlik bilgisini denetleyemiyor. Bunu telafi edecek şekilde bir adım daha ileri gideceğiz şimdi...

Tanım: Azaltma kuralı: Eğer j , i düğümünden köke olan yol üzerinde bir düğüm ise (i kök değil) j düğümü doğrudan köke bağlanır.

Bunun graf teorideki karşılığı “kaplama” ya da “kapama”. Graf teori ödevlerinden birinde, ben söylemeyeceğim, ama siz bunu kullanmalısınız. Problem farklı, alt yapısı bu.



```
1: int Kume_bul_2( int i ){
2:   int r, s;
3:   for( r = i ; ata[r] >= 0 ; r = ata[r] ); //kökü bul (ata bilgisi negatif değer)
4:   while( i != r ){ // kökte miyim (köke bağlı bir düğüm için de fazladan çalışır)
5:     s = ata[i]; // i'nin atasını sakla
6:     ata[i] = r; // i düğümünü doğrudan köke bağla
7:     i = s; // i'nin atasına geç
8:   } return r; }
```

“*For vardı, bir de while döngüsü ekledik. Bu nasıl iyileştirme, anlamadım!*” Ben de anlamadım, bakalım. Küme_bul_2(7) ikinci kez çağrılınca ve sonraki çağrılarda bir çırpıda $r=0$ bulacaktır.

Lemma: (Tarjan ve Jan van Leeuwen)

Her biri **tek bir düğüme sahip** kümeler ile başlanıldığında **f** adet küme bulma ve **u** adet küme birleştirme işlemleri her hangi bir karışık biçimde uygulandığında **T(f, u)** gereksinim duyulacak maksimum zaman (adım) olarak tanımlansın. **u ≥ n/2** kabul edildiğinde (n, eleman sayısı);

$$k_1 \cdot (n + f \cdot \alpha(f + n, n)) \leq T(f, u) \leq k_2 \cdot (n + f \cdot \alpha(f + n, n)) \text{ verilir. } (k_1, k_2 \text{ pozitif sabitler})$$

Buradaki $\alpha(f, u)$ Ackermann fonksiyonunun tersidir, çok yavaş artan bir fonksiyondur.

Ackermann fonksiyonu: (Final sorusu: Ackermann fonksiyonunu özyineli olarak yazınız)

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Animated Ackermann's function calculator:

<http://gfredericks.com/math/Arith/ackermannPage.html>

Ackermann fonksiyonu, çok hızlı büyüyen bir fonksiyondur. Ör: $A(5, 1) = 2^{2^{2^{2^2}}} - 3$ (65536 tane 2 rakamı birbirinin üssü) şeklinde yazılır. Ackermann fonksiyonunun tersi olan **ters Ackermann fonksiyonu** (*inverse Ackermann function*) ise çok yavaş büyüyen bir fonksiyondur. $m \geq n \geq 1$ olmak üzere şöyle tanımlanır:

$$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n\}.$$

$A(3, 4) = 16$, $q < 2^{16} = 65536$ ve $p \geq q$ için $\alpha(p, q) \leq 3$ (yani 3 adım sonra sonuca ulaşırsınız)

$q = n$ eleman sayısı, $p = n + f$ alındığında $A(4, 1)$ çok çok büyük bir değer olması nedeniyle **uygulamada $\alpha(p, q) \leq 4$ olacaktır.**

n 'e bölünce işlem başına maliyet $O(1)$ 'e düşer, maliyet hızla azalır. Ne kadar azalırın analizi de budur. Düşünün; 1 milyon eleman var, ben $O(1)$ 'de buluyorum, sen daha arıyorsun listede... Hani dedik ya, aynı işe girecek olan matematikçi ile mühendisin farkı budur işte!

Tarjan ünlü bir graf teorcidir, birçok teorisi var. Leeuwen onun doktora öğrencisi olmalı, bu lemma da bir doktora çalışması olabilir.

Keşke makale yazabilir hale getirilseniz... Lisansta 4 yıl okuyup bir meslek kazanıyorsunuz, doktora da ise sadece bir problem üzerine 4 yıl çalışıyorsunuz. Dünyada çözülememiş bir problemi çözüyorsunuz, ancak 4 yıl yoğunlaşıncaya çözebiliyorsunuz. Her adam da 4 yılda çözebilir mi? Yok! O zaman herkes doktor olurdu. İlk başta hoca, "Sen şuna başla!" der; hoca da, öğrenci de o problemin çözülüp çözülemeyeceğini bilmez. Uğraşırsın, bir şey çıkarsa çıkar; belki çıkmayabilir de... O yüzden grup içinde çalışırsan, sürekli öğrenirsin.

Mustafa Hoca'nın tezi: *Reliability Analysis in Fault-Tolerant Software and An Error Recovery Study*. (Hataya Hoş Görülü Yazılım Geliştirme)

Bu konuları bir haftada bitiremezsiniz, hiç çalışmayın bitmez. Her gün düzenli olarak 2-3 saat çalışmanız lazım. O haftanın konusunu hemen o hafta tekrar etmelisiniz.

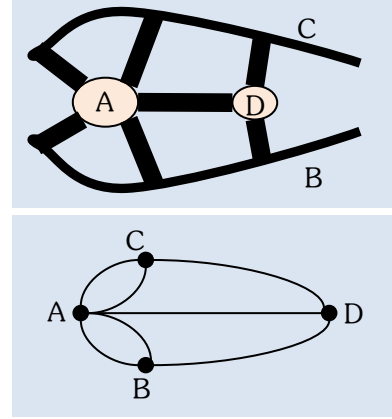
ÇİZGELER (GRAPHS)

Çizge kuramının başlangıcı Königsberg'in 7 köprüsü (*Kaliningrad / Rusya*) problemine dayanır. 1736'da Leonhard Euler'ın söz konusu probleme ilişkin kullandığı sembol ve çizimler çizge kuramına ilişkin başlangıç olarak kabul edilir. [\[graph data structure\]](#) [\[graph theory\]](#)

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?

Euler (yûlir okunur) kara parçalarını düğüm, köprüleri de kenar kabul eden bir çizge elde etti (Notasyon/çizim farklılığı). Söz konusu yürüyüşün ancak her düğümün derecesinin çift olması durumunda yapılabileceğini gösterdi. (Teorem bu, ispatını sonraki derslere bırakıyoruz.)

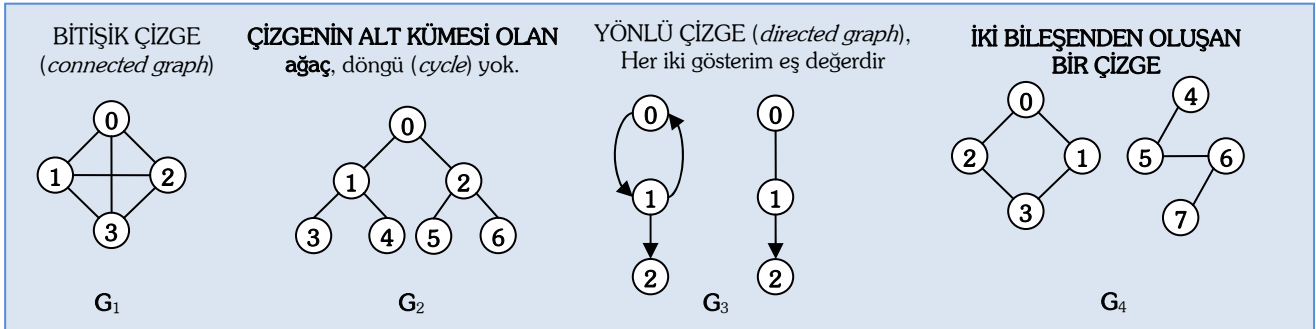
Çizge kuramının bu ve buna benzer problemlerle başladığı düşünülür. 4. Sınıfta “Çizge Kuramı ve kombinatorikler”, lisansüstünde ise “Çizge Kuramı” dersleri var. Günlük yaşamdaki birçok problemi çizge kuramı uzayına taşıyarak çözebiliyoruz.



TANIMLAR

Çizge: G çizgesi boş olmayan iki sonlu (*finite*) kümeden meydana gelir. $G = (V, E)$ biçiminde sembolize edilir. V , düğümler (*vertex-node*); E , kenarlar (*edge-arc*) kümesidir. Yönsüz çizgede kenarları temsil eden düğüm ikilileri sınırsızdır. Alışageldiğimiz parantez kullanılarak $(v_0, v_1) \equiv (v_1, v_0)$ aynı kenarı gösterir. Yönlü çizgede ise (*directed graph - digraph*) ise $\langle v_0, v_1 \rangle$ ve $\langle v_1, v_0 \rangle$ iki ayrı kenarı gösterir ($\langle \text{tail}, \text{head} \rangle$: *tail*, okun başladığı; *head* ise sivri ucunun bulunduğu taraftır).

Bu tanıma göre çizge, kartezyen çarpımın bir alt kümesidir. Çünkü tüm ikilileri barındırmaz.



$$V(G_1) = \{0, 1, 2, 3\}$$

$$E(G_1) = \{(0,1), (0,2), (0,3), (1,2), (1,3), (2,3)\}$$

$$V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$$

$$E(G_2) = \{(0,1), (0,2), (1,3), (1,4), (2,5), (2,6)\}$$

$$V(G_3) = \{0, 1, 2\}$$

$$E(G_3) = \{\langle 0,1 \rangle, \langle 1,0 \rangle, \langle 1,2 \rangle\}$$

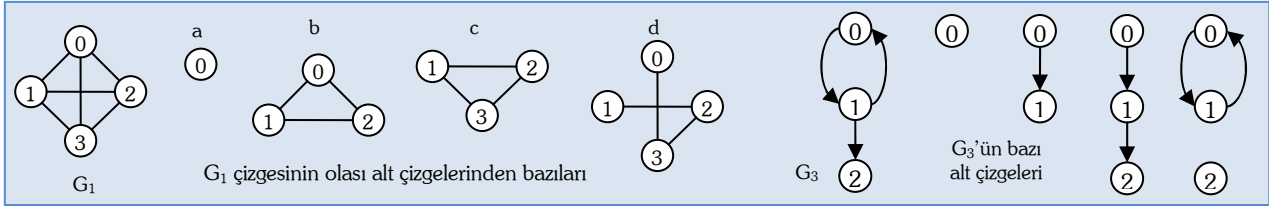
ÇİZGE VERİ YAPISI NERELERDE KULLANILIR?

Nerelerde kullanılmaz ki! Çizgelerin çok geniş bir alanda kullanımı söz konusudur. Elektrik devrelerinin analizi, proje planlama, kimyasal bileşenleri tanıma... Bilgisayar bilimlerinde; *shortest path* (en kısa yol), *connected graph* (birleşik çizge), *minimum spanning tree*, *activity network*, *trees* (ağaçlar), *sorting* (sıralama), *finite automoto* (sonlu özdevinirler), *networks* (ağla ilgili birçok problem), *flows in transport network* (akış, taşıma), *max flow* (kaynaktan hedefe en fazla petrol/su taşıma problemi).

TERMİNOLOJİ

Komşu düğümler: (v_0, v_1) kenarı yönsüz çizgede bir kenar ise v_0 ve v_1 düğümlerine **komşu düğümler** (*adjacent*) adı verilir.

Alt çizge: G' çizgesi, G çizgesinin alt çizgesi ise $V(G') \subseteq V(G)$ ve $E(G') \subseteq E(G)$ olur.



Yol: Yönsüz bir çizgede v_p 'den v_q 'ya olan bir yol (*path*) $(v_p, v_{i1}), (v_{i1}, v_{i2}), \dots, (v_{in}, v_q)$ 'nın kenar olduğu $v_p, v_{i1}, v_{i2}, \dots, v_{in}, v_q$ düğümler dizisidir. Eğer yönsüz bir G çizgesinde v_0 'dan v_1 'e bir yol var ise v_0 ve v_1 düğümleri bitişik (*connected*) olarak isimlendirilir. G çizgesinde tüm farklı düğüm ikilileri (v_i, v_j) ($i \neq j$) arasında bir yol var ise G çizgesinin bitişik olduğu söylenebilir. Bir düğümü uç kabul eden düğümlerin sayısı söz konusu düğümün derecesini meydana getirir. G_2 çizgesinde 0 düğümünün derecesi 2'dir. G_1 'de $d(V_1)=3$, G_2 'de $d(V_4)=1$ 'dir. Yönlü çizgede ise düğümün derecesi giriş (*in-degree*) ve çıkış (*out-degree*) değerlerinin toplamıdır. G_3 çizgesinde $d=1+1=2$ bulunur.

Teorem 1: Eğer bir G çizgesinde n adet düğüm, e adet kenar var ise **düğümlerin derecesi toplamı, kenarların sayısının iki katına** eşittir: $2e = \sum_{i=1}^n d_i$

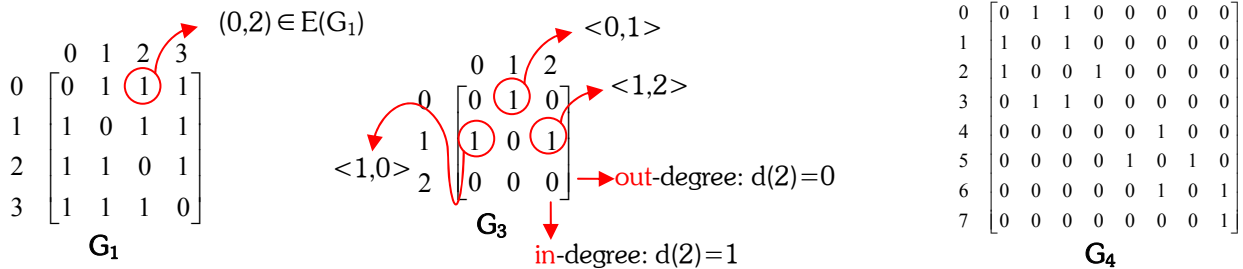
İspatı: Her kenarın, kendi uçlarında bulunan düğümlerin toplam derecelerine yaptığı katkı 2'dir. Hem basit olan, hem de basit olmayan çizgelerde bu geçerlidir. **Basit çizge:** İki düğümü birleştiren en fazla bir kenar olabilen ve düğümleri kendine döngü (*self-loop*) içermeyen çizgedir.

Önce kavramı anlatıyoruz, sonra bilgisayar ortamında gerçekleştirilmesini... *Discrete Mathematics* dersini aldınız, bunları görmüş olmalısınız. Derste öğretilmemişse bile, siz kitabını alıp okumalıydınız. "Para kazanacağın kitabı oku!" diyor hoca, "Yok, almıycam!" diyorsun. Sen bilirsin. Lütfen mesleğinizle ilgili kitaplar okuma alışkanlığını bu yaşlardayken kazanın. Bunları okuyun. Yazı çok iyi değerlendirin. Eksiğiniz neyse onu tamamlayın. Burada amacımız *discrete mathematics* anlatmak değil. 'Grafı bellekte nasıl gerçekleştiririz?'i anlatmak.

ÇİZGE VERİ YAPISININ GERÇEKLEŞTİRİMİ

1- Komşuluk Matrisi (*Adjacency Matrix*)

$G(V, E)$, n düğümlü bir çizge olsun. $n \times n$ boyutunda bir matris ile bir G çizgesi bellek ortamında gerçekleştirilebilir. Yönsüz bir çizgede 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 $O(n^2)$ olur.



Derece bulma: Matrisler gerçekleştirimde bir düğümün derecesini bulmak için ilgili satır ya da sütundaki 1'ler sayılmalıdır ($O(n)$). Eğer çizge yönlü ise satırdaki 1'ler *out-degree*, sütundakiler ise *in-degree* değerini verir. Bu ikisinin toplamı düğümün derecesidir ($O(2n)$).

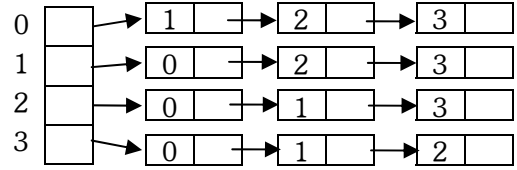
Kenar sayısı bulma: Kenar sayısını bulmak için tüm matristeki 1'leri sayıp 2'ye böleriz ($O(n^2)$). Yönsüz çizgede matris simetrik olacağı için yarısına bakmamız yeterli olur, yine de karmaşıklık n^2 düzeyindedir ($O(n^2/2) = O(n^2)$).

Bileşen sayısı bulma nasıl yapılır? Düşünün.

2- Komşuluk Listesi (Adjacency List)

Çizgeyi matris olarak gerçekleştirmek yerine matristeki her satır için liste veri yapısı tercih edilir. Bu yapının dezavantajı her kenar için 2 tane öge kullanıyor olmamızdır.

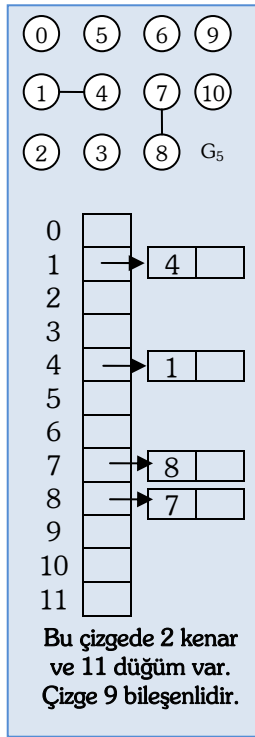
```
1: #define MAX_DUGUM 50 /* max düğüm sayısı */
2: typedef struct oge *oge_gost;
3: typedef struct oge {
4:     int dugum;
5:     oge_gost bag;
6: }
7: oge_gost baslik[MAX_DUGUM];
```



G₁'in komşuluk listesiyle gösterimi

Kenar sayısı bulma: Bu yapıda (n düğüm, e kenar varsa) “toplam kenarların sayısı” bulmanın karmaşığı $O(n+e)$ olacaktır. (Çizge kuramında karmaşıklık genellikle e 'ye bağlı verilir.) Listedeki tüm kenarları sayıp 2'ye bölünce toplam kenar sayısını bulacağımızı Teorem 1 garanti eder.

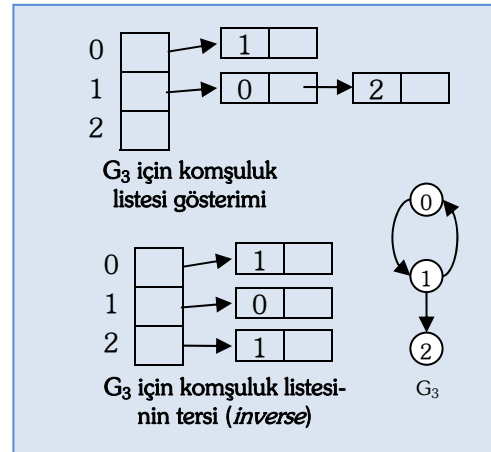
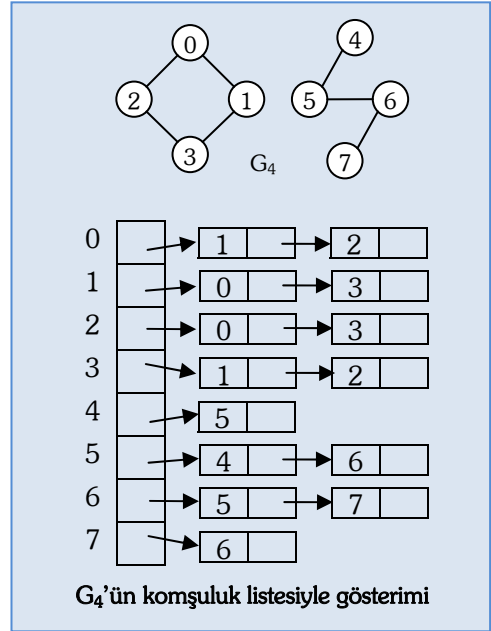
Yönsüz bir çizgede (Ör. G_1) kenar sayısını daha hızlı bulmak için yapının sıralı olması avantajını kullanabiliriz. Listelerin başlarından itibaren kuyruk değeri baş değerinden küçük olan ((0, 1), (1, 2) gibi) kenarlara kadar sayıp, bu kenarları saymadan bir sonraki düğüme geçebiliriz. Böylece düğümlerin yarısını dolaşarak kenar sayısını bulabiliriz. Ama karmaşıklık yine de $O(n+e)$ düzeyinde kalacaktır.



Yönlü çizgelerde ise (Ör. G_3) anlatılan ‘sıralı olma avantajı’ kullanılamaz. Çünkü $\langle 0,1 \rangle$ ve $\langle 1,0 \rangle$ kenarları birbirinden farklıdır. Bu yüzden tüm kenarlar sayılmalıdır.

Derece bulma: Yönsüz çizgede (Ör. G_4) bir düğümün derecesini bulmak için ilgili başlıktaki (i . başlık) öğeleri saymak yeterli olacaktır.

Yönlü çizgede ise (Ör. G_3) düğümün derecesi, gelen ve çıkan kenarların toplamıdır. Bu yapı, söz gelimi, 1 düğümünden kaç düğüme gidildiğini (*out-degree*) saymamıza olanak sağlar. Fakat 1 düğüme kaç düğümden gelinebildiğini (*in-degree*) bulabilmek için listelerin tamamını dolaşmamız gerekecektir. Öyleyse yönlü çizge için **bu yapıyla beraber bu listenin tersini de tutmalıyız** ki, hem gelen hem de çıkan dereceleri hızlıca bulabilelim. Eğer tutmazsak, dereceleri bulmak için listelerin hepsini dolaşmamız gerekecektir. **NOT:** Bu liste bir kez dolaşarak devrik kurulur.



3- Dizi ile Sıralı (Sequential) Gösterim

n adet düğüm ve e adet kenarın bulunduğu **yönsüz bir çizgeyi** dizi kullanarak gerçekleştirebilir miyiz? Evet! Her düğümün komşuları bir liste gibi düşünülebilir. $\text{Düğüm}[i]$, $0 \leq i < n$ değerleri i düğüme komşu olan listelerin başlangıç adresini tutmak için kullanılır. Her kenarın düğüm dereceleri toplamına yaptığı katkının 2 olduğu hatırlanırsa toplam listeler için $2e$ adet düğümün yer alması gerekir. $\text{Düğüm}[n] \leftarrow n+2e+1$ olacaktır. i . düğüme komşu olan düğümler listesi $\text{düğüm}[i]$, ..., $\text{düğüm}[i+1]$, $0 \leq i < n$ indisli düğüm elemanlarıdır.

G4 çizgesinin örnek bir gerçekleştirimi şu şekildedir: (tanımlanacak dizi: $\text{int } \text{dugum}[n + 2 \cdot e + 1]$)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	1	2	0	3	0	3	1	2	5	4	6	5	7	6
n								1	$2 \cdot e$													

Ör 1: 0 nolu düğümün komşuları $[0]$ 'da bulunan 9'dan (9 dâhil) $[1]$ 'de bulunan 11'e (11 hariç) kadar, yani $[9]$ ve $[10]$ indislerinde bulunan düğümlerdir: 1, 2.

Ör 2: 1'in komşuları $[1]=11$ ve $[2]=13$ olduğu için $[11]-[12]$ indislerinde yer alan 0 ve 3'tür.

Ör 3: 7'nin tek komşusu ise $[7] = 22$ ve $[8] = 23$ olduğundan $[22]-[22]$ indislerinde bulunan 6 nolu düğümdür.

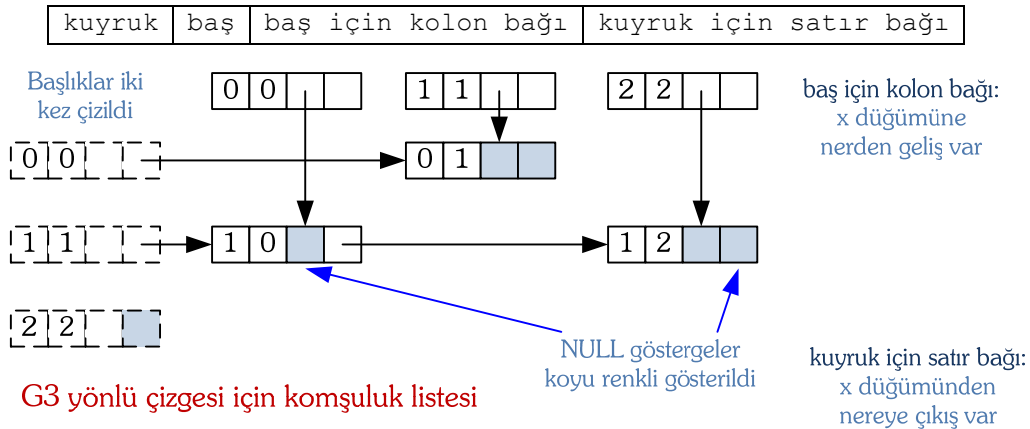
Elbette bu dizinin içeriği, gerek verilerin geliş sırasına gerekse de programcının diziyi oluşturan yordamı yazış biçimine bağlı olarak farklı şekillerde oluşabilir.

Listelerde bulunan **2e adet öge alma** dezavantajı bu yapıda yoktur. Fakat bu yapı da, komşuluk listesinin sağladığı avantajı, esnekliği sağlamaz.

Yorum: Depolamak ayrı konudur, o veri üzerinde çalışacak algoritmaların etkinliği ayrı konudur. Bu yapı için konuşacak olursak; aramada sorun yok, ama ekleme ve silme işlemlerinde esneklik sağlıyor mu? Hayır! Çok sayıda ekleme-silme işlemi için bu yapı verimli değildir. Arka plandaki problemi bilemediğimiz için *"Bu yapılar arasından şunu tercih ederiz."* diyemiyoruz. Bu şekilde de gerçekleştirebiliriz. Örneğin bir ağdaki bilgisayarlar sabittir. Problem, onları tutmaksa bu yapı uygun olabilir.

4- Yönlü Çizgelerin Komşuluk Listesi (Orthogonal) Gösterimi

Yönlü çizgeler için alternatif düğüm yapısı şu şekilde de olabilir.



Satır başlıkları ile sütun başlıkları aynı öğelerdir. Çizimi basitleştirmek için başlıklar ikişer kez çizildi. Baş için kolon bağı diye isimlendirdiğimiz gösterge, bir düğüme nerden geliş olduğunu gösteriyor. Kuyruk için satır bağı diye isimlendirdiğimiz gösterge ise bir düğümden nereye çıkış bulunduğunu gösteriyor. Görüldüğü gibi **her bir kenar ögesi, iki ayrı listede yer alıyor**. Aynı kenar için iki öge almamız gerekmiyor, tek öge almak yeterli oluyor.

Bu yapı, Veri Yapıları 1 dersinde gördüğümüz **seyrek matris yapısına** benziyor. Bu yapıyı da onun gibi döngülü, çift bağlı ve başlıklı hale getirebilirsiniz. Ayrıca diğer başlığa geçmek için ve tür için yeni bağ alanları da eklenebilir.

2e adet eleman almak rahatsız etti, böylece bu örneği bulduk. Şimdi bir örnek daha...

Aklınıza gelen ilk çözüme hiç güvenmeyin, o koda da güvenmeyin.
Arka plandaki veri yapısı bu değil de başka bir şey olsaydı, daha verimli çözebilir miydiniz?

5- Komşuluğun Çok Bağlılı Listeler ile Gösterimi (Adjacency Multilist)

Bu gerçekleştirim türünde kenarları temel öge olarak ele alıyoruz. Yani; (v_i, v_j) kenarı temel ögedir, **bir kenar için yalnızca bir öğemiz olacaktır**. Fakat bu öge, kendisini meydana getiren ve uçları olan her iki düğümün (hem v_i 'nin ve hem de v_j 'nin) komşuluk listesinde yer almalıdır.

Bir de herhangi bir kenar işlenmişse, üzerinden geçilmişse bu kenarın tekrar ele alınmasını önleyecek bir işarete gereksinim vardır. Dolayısıyla komşuluğun çok bağlılı liste ile gösteriminde kullanılacak veri yapısı **işaret** adı verilen ek bir veri alanını içermelidir.

isaret	dugum1	dugum2	list1	list2
--------	--------	--------	-------	-------

```

1: typedef struct kenar *kenar_gost;
2: typedef struct kenar {
3:     short int isaret;
4:     int dugum1;
5:     int dugum2;
6:     char veri;
7:     kenar_gost list1;
8:     kenar_gost list2;
9: }
10: kenar_gost baslik[MAX_DUGUM];

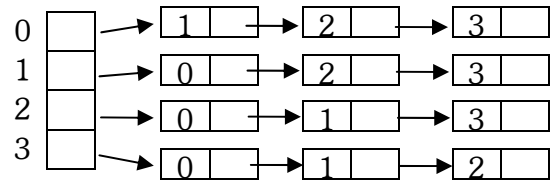
```

Dolaşmak: Bir kenar ögesinin `dugum1` alanında söz gelimi 2 değeri varsa, `dugum1` alanında 2 olan diğer düğümlere ulaşmak için `list1` bağıni takip ederiz. Aynı şekilde `dugum2` alanında söz gelimi 1 varsa `dugum2` alanında 1 olan diğer öğelere `list2` bağıni takip ederek ulaşırız. NULL değeriyle karşılaştığımızda artık aranan özellikte hiçbir öge kalmamıştır. **Başka var mı? Bağı takip et!**

Ekleme: Çok bağlılı liste ile tutulan çizgeye bir kenar eklemek için ilgili listede NULL bulana kadar ilerleyip yeni kenarı sona (NULL bulduğumuz yere) ekleriz. Fakat başa eklememiz de mümkündür, böylece liste sonuna kadar dolaşma maliyetinden kurtulmuş oluruz.

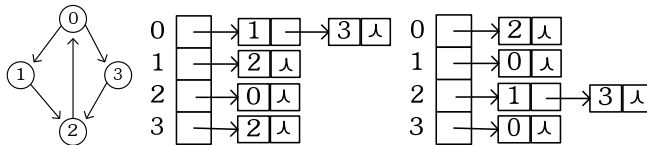
Listelerin başlangıçlarına gösterge tutan başlık dizisinde hiç bağlantısı olmayan bir öge (*isolated vertex*) NULL değerini içerir.

Yönsüz çizgeler için komşuluk listesi gösteriminde bir baş dizisi tutuyorduk. Yönlü çizgeler için ise baş dizisiyle birlikte bir de kuyruk dizisi tutulmalıdır.

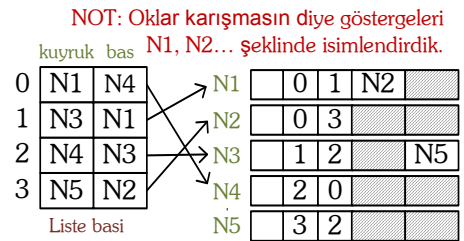


G₁'in komşuluk listesiyle gösterimi

Alıştırma: Kenar listesi verilen yönlü çizge için komşuluk matrisini, komşuluk listesini ve komşuluk çok bağlılı listesini veriniz: $\langle 0,1 \rangle$, $\langle 0,3 \rangle$, $\langle 1,2 \rangle$, $\langle 2,0 \rangle$, $\langle 3,2 \rangle$.



Üstte görüldüğü üzere, **yönlü bir çizgeyi komşuluk listesiyle göstermek için** biri gelen kenarlar için, diğeri (listenin devriği) çıkan kenarlar için olmak üzere **iki tane liste tutuyorduk**. Böylece kenar sayısının 2 katı kadar (**2e adet**) öge kullanmamız gerekiyordu. Eğer çizgeyi çok bağlılı liste kullanarak gösterirsek yandaki gibi sadece **kenar sayısı kadar öge almamız yeterli olur**.



Bu sırayla gelip “O(1) de yerleşsin” denilirse liste başına eklenecek! Böyle (liste sonuna ekleyerek) çizene puan vermem. Çünkü liste sonuna eklediğini görüyorum.

Çıkarım: Aklınıza gelen ilk çözüm, iyi bir yöntem olmayabilir. Belki n^3 ’lük bir algoritmayı n ’e indirmeniz mümkün olabilir. Bu noktada veri yapısını iyi tasarlamak gerekir. Daha kod yazmadan önce veriyi nasıl depolayacağımı düşünüyorum. Veriyi depolamak o kadar kolay değil!

Çizge gerçekleştirimiyle ilgili anlatacaklarımız burada bitti. Bundan sonra işlemler üzerinde duracağız.

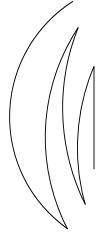
ÇİZGE ÜZERİNDE TEMEL İŞLEMLER

Ağaçlar üzerinde ağaç dolaşımları (kök önde, kök sonda, kök ortada, düzey düzey) sıkça kullanılan işlemler olduğu gibi çizgelerde de benzer bir durum vardır. Yönsüz bir çizge $G=(V,E)$ verildiğinde bir V düğümünden olası tüm düğümlere $[V(G)]$ uğramak (ziyaret etmek) isteyebiliriz. Bu işlemi iki farklı biçimde yapabiliriz.

1. Önce derinliğine arama (*depth first search*)
2. Önce genişliğine arama (*breadth first search*)

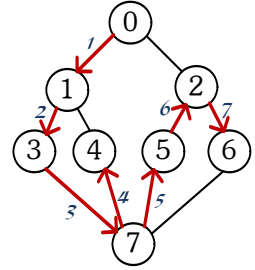
Önce derinliğine arama kök önde dolaşıma, önce genişliğine arama ise ağaçta düzey düzey dolaşıma benzer. Her iki arama için çizgenin komşuluk listesi ile gerçekleştirildiğini kabul edeceğiz.

Önce Derinliğine Arama (Depth First Search)

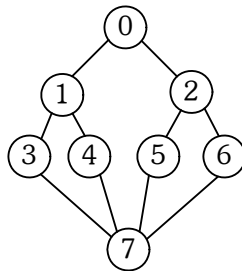


Başlangıç düğümü v ziyaret edilerek arama başlar. 'Ziyaret etme', ilgili düğümün veri alanının yazdırılması anlamındadır. Daha sonra v düğümünün komşuluk listesinden ziyaret edilmemiş bir w düğümü seçilir. Önce derinliğine arama, w düğümü üzerinde devam eder. [\[wiki\]](#)

v 'nin komşuluk listesinde hali hazırdaki konumumuzu saklamak üzere **yığıt** kullanırız. Sonuçta bir u düğümü üzerinde iken ziyaret edilecek bir düğüm kalmadığında yığıttan bir düğüm alınır. Bu düğümün komşuluk listesi üzerinden süreç devam ettirilir. Karmaşık gibi görünen çözümleme, **özyineli yordam** kullanıldığında kolaylaşacaktır. Yordamda global bir dizi kullanılacaktır.

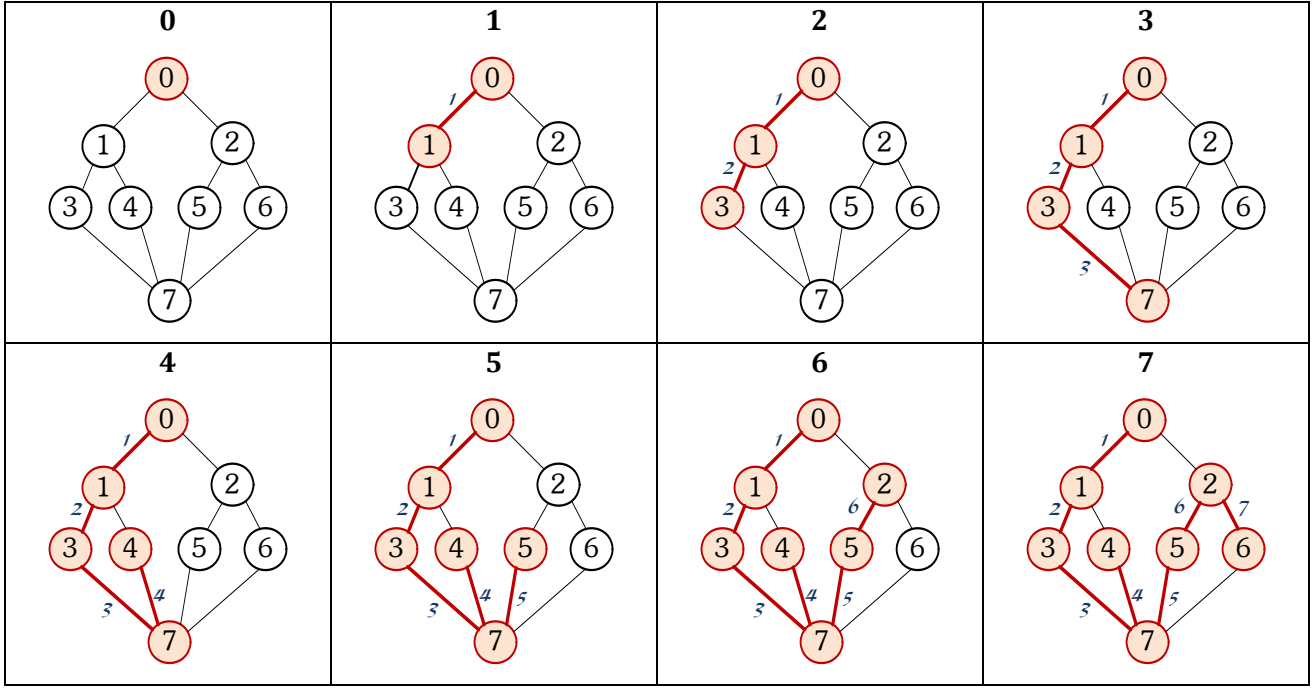


Ekleme: Kenarlar geliyor, yapıya ekleyebilmelisin. Ben bakıp çizdim, sen bakıp çizemezsin ki! Senin algoritman çalışacak. (ekleyen-kuran yordam?)



başlık				
0	1	2		
1	0	3	4	
2	0	5	6	
3	1	7		
4	1	7		
5	2	7		
6	2	7		
7	3	4	5	6

```
1: #define FALSE 0
2: #define TRUE 1
3:
4: short int ugranildimi[MAX_DUGUM]
5:
6: typedef struct oge* oge_gost;
7: typedef struct oge {
8:     int dugum;
9:     oge_gost bag;
10: };
11: oge_gost baslik[ MAX_DUGUM ];
12:
13: void OnceDerinligineAra( int v ){
14:     // arama (ziyaret) v düğümünden başlar
15:     oge_gost w;
16:     ugranildimi[v] = TRUE; // uğradım bilgisini koy
17:     printf("%5d ", v);
18:     for( w = baslik[v] ; w ; w = w->bag )
19:         if( !ugranildimi[ w->dugum ] )
20:             OnceDerinligineAra( w->dugum );
21: }
```



Dolaşma sırası: 0, 1, 3, 7, 4, 5, 2, 6. Dikkat edin! Dolaşım sonucunda bir ağaç oluştu. Bu ağaca bir kenar daha eklersek döngü oluşur ve ağaç, ağaç olmaktan çıkar.

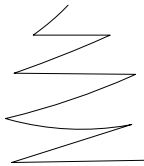
Başlangıç düğümü: Bu çizgede eğer 0 düğümünden başlayarak önce derinliğine arama yaparsak sırasıyla 0, 1, 3, 7, 4, 5, 2, 6 düğümleri ziyaret edilecektir. Başlangıç düğümü 0 olmak zorunda değildir, elbette. Her hangi başka bir düğüm, 6, 4 vs. de olabilirdi. Biz olayı daha iyi görebilmek için 0 düğümünden başladık. Başka bir düğümünden başlasaydık aynı öğeler farklı bir dizilişte karşımıza çıkacaktı.

Listenin sıraya etkisi: Çizgeyi bu şekilde komşuluk listesiyle gerçekleştirdiğimizi kabul ettik. Verilerin kullanılacak kuyruğun önüne mi, yoksa sonuna mı ekleneceği; listenin sıralı olup olmayacağı size kalmış... Fakat bu listenin kurulma biçimi, dolaşma biçimini etkileyeceği için önemlidir. Farklı biçimde kurarsak farklı sırada dolaşırız. Örneğin 0'ın komşuluk listesi {1, 2} sırasında değil de {2, 1} sırasında olsaydı dolaşım sırası da değişecekti.

Analiz: Önce derinliğine arama yordamı her düğüm için en fazla 1 kez işletilir. Eğer çizgede n tane düğüm, e tane kenar var ise algoritma karmaşıklığı $O(e+n)$ olur. Kenar sayısının en fazla $n \cdot (n-1)/2$ olabileceğini belirtelim.

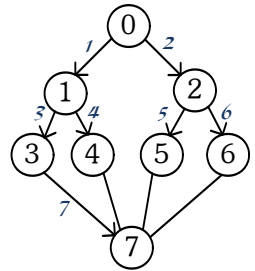
Veri yapısı: Sana verilen problemi çözdün, analiz ettin ve bu yapıyı uygun buldun. Eğer G çizgesi komşuluk listesiyle değil de komşuluk matrisiyle gösterilseydi, v düğümüne komşu olan tüm düğümleri belirlemenin karmaşıklığı $O(n)$ olacağından, arama yordamı n adet düğüm için $O(n^2)$ 'lik bir zamana gereksinim duyacaktı. Bu şekilde ise lineer zamanda dolaşabiliyoruz.

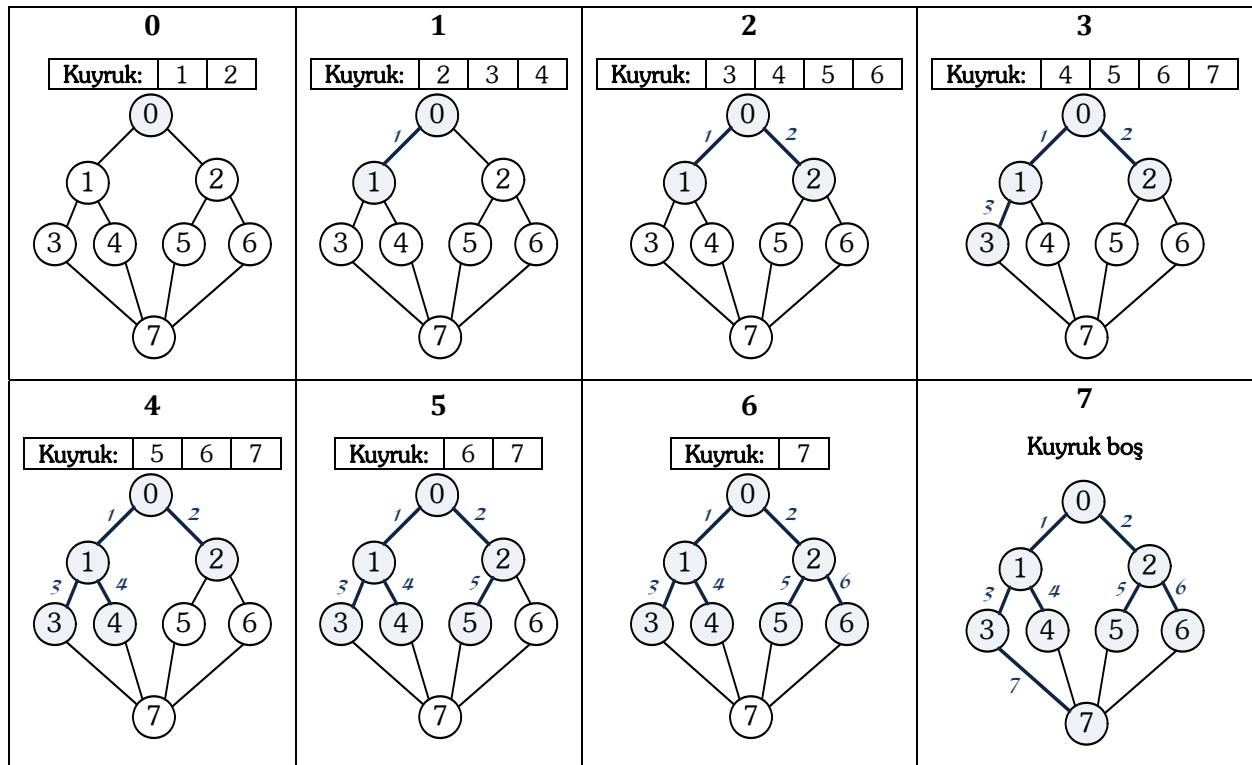
Önce Genişliğine Arama (Breadth First Search)



Önce genişliğine arama v düğümünden başlar, bu düğüm ziyaret edilmiş olarak işaretlenir. Daha sonra v düğümünün komşuluk listesinde yer alan düğümlerin hepsi ziyaret edilir. Söz konusu listedekiler ziyaret edildikten sonra v 'nin komşuluk listesinde yer alan ilk düğüme komşu olan düğümlerden ziyaret edilmemiş olanların hepsi ziyaret edilir. Bu mantığı gerçekleştirmek için her

bir düğümü ziyaret ettiğimizde düğümü kuyruğa, ilk atılanın ilk alınması gerektiğini söyleyen kuyruk yapısına, atarız. Bir düğüme ilişkin komşuluk listesinin sonuna gelindiğinde kuyruktan bir düğüm alır ve bu düğümün komşuluk listesinde ilerleriz. Bu şekilde devam edilerek ziyaret edilmemiş düğümler ziyaret edilir ve kuyruğa eklenir. Kuyruk boşaldığında aramaya son verilir. Nasıl ki, önce derinliğine aramada problemin doğası gereği yığıt kullanma zorunluluğu vardı; bunda da kuyruk kullanma zorunluluğu vardır. Anlatım tarzından kuyruk veri yapısı kullanılması gerektiği çıkıyor. [\[wiki\]](#)





```

1: typedef struct kuyruk *kuyruk_gost;
2: typedef struct kuyruk {
3:     int dugum;
4:     kuyruk_gost bag;
5: };
6:
7: void kuyruk_ekle( kuyruk_gost *, kuyruk_gost *, int );
8: int kuyruk_al( kuyruk_gost * );
9:
10: void OnceGenisligineAra( int v ){
11:     // v düğümünden itibaren önce genişliğine aramaya başlar
12:     // genel dizi ugranildimi[] başlangıç değerleri FALSE olmalıdır
13:     // dinamik kuyruk veri yapısı kullanılır
14:     oge_gost w;
15:     kuyruk_gost on, arka;
16:     on = arka = NULL;
17:     printf("%5d ", v);
18:     ugranildimi[v] = TRUE;
19:     kuyruk_ekle( &on, &arka, v );
20:     while( on ){
21:         v = kuyruk_al( &on );
22:         for( w = baslik[v] ; w ; w = w->bag )
23:             if( !ugranildimi[ w->dugum ] ){
24:                 printf("%5d ", w->dugum);
25:                 kuyruk_ekle( &on, &arka, w->dugum );
26:                 ugranildimi[ w->dugum ] = TRUE;
27:             }
28:     }
29: }

```

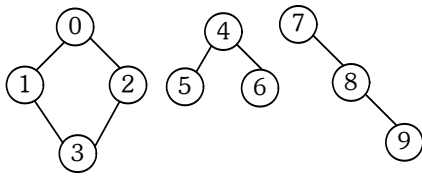
Analiz: Her düğüm kuyruğa 1 kez ekleneceğinden dolayı while döngüsü n (düğüm sayısı) kez işletilir. Komşuluk listesi gösteriminde $2e$ adet kenar bilgisi olduğundan ötürü algoritma karmaşıklığı $d_0 + d_1 + d_2 + \dots d_n = 2e = O(e)$ olur. **$O(n+e)$ olur.** Komşuluk matrisi kullanılsaydı $O(n^2)$ olacaktı.

İlerisi: Önce derinliğine/genişliğine arama kavramları uzayı ağaca benzetip dolaşma terminolojisinde; Kesikli Matematik'te *min-max*, *b* search algorithm*, *branch and bound* konularında geçer. Oyun problemlerinde de eğer "Hamlelerin şunlar olabilir." diyorsan, genişliğine açıyorsun demektir.

Bitişik Bileşenler (Connected Components)

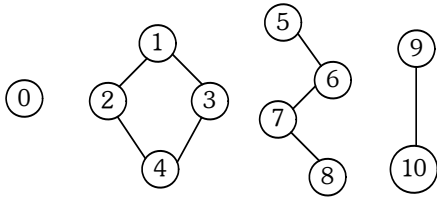
Önce derinliğine ve genişliğine arama algoritmaları temel algoritmalar olup çizge üzerinde değişik problemlerin çözümünde kullanılırlar. Yönsüz bir çizgenin **bitişik olup olmadığı** ya da **kaç bileşenden oluştuğu** konusuyla ilgilenelim. **Aşağıdaki yordam derinliğine arama algoritmasını kullanarak bir çizgenin bileşenlerini yazdırır.** Yeni bir bileşene geçildiğinde yeni bir satıra geçer. Böylece her bir bileşenin hangi düğümlerden oluştuğunu bulmuş olur. Her yeni satıra geçildiğinde artan basit bir değişken koyarak **bileşen sayısını** da bulabiliriz. [\[wiki\]](#)

```
1: void bitisik_bilesenler( void ){
2:     // bu yordam, çizgenin bileşenlerini ortaya çıkartır
3:     for( int i = 0 ; i < n ; i++ )
4:         if( !ugranildimi[i] ){
5:             OnceDerinligineAra(i);
6:             printf("\n");
7:         }
8: }
```



Çıktı:

```
0 1 3 2
4 5 6
7 8 9
```



Çıktı:

```
0
1 2 4 5
5 6 7 8
9 10
```

Analiz: Dıştaki for döngüsü n kez dönüyor, içteki OnceDerinligineAra() yordamının karmaşıklığı $O(n+e)$ idi. Karmaşıklık ikisinin çarpımı kadar mı olur? $O(n^2)$ mi olur? Olur mu? Olur mu? Olur mu? O kadar çalışır mı?

Hayır! Tek bileşen varsa ilk döngüde $O(e+n)$ kez çalışır ve ugranildimi[] dizisindeki tüm değerler TRUE olur. Bu yüzden i'nin alacağı diğer tüm değerlerde if bloğuna girilmeden sonraki döngüye geçer. Toplamda **$O(e+n)$** kez çalışır. “Hepsini çarparım, sonucu söylerim.” Değil!

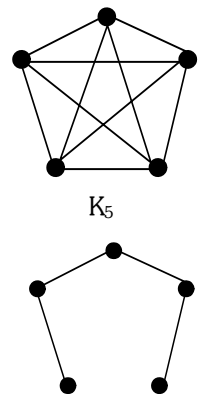
Eğer arka plandaki veri yapımız liste yerine matris olsaydı, bileşen sayısını bulma karmaşıklığı $O(n^2)$ olacaktı. O da etkili yazarsan...

Teorem: En az $(n-1)$ kenar bulunmalıdır ki, çizge birleşik olsun.

Tek yönlü bir teoremdir; $n-1$ kenar bulunması gereklidir, ama yeterli değildir. Yani $n-1$ kenarı olan bir çizge birleşik olabilir de olmayabilir de. Fakat $n-1$ 'den daha az kenar varsa çizge kesinlikle bitişik olamaz.

Bir çizgede en fazla kaç kenar bulunabilir? Çizgedeki tüm düğümler diğer tüm düğümlerle komşuysa kenar sayısı şöyle bulunur; $(n-1)+(n-2)+\dots+2+1 = n \cdot (n-1)/2$ Böylece e'nin n^2 'li bir ifade olabileceğini de görmüş oluyoruz. Öyleyse bitişik bir çizgenin kenar sayısı şu aralıkta bulunabilir:

$$n-1 \leq e \leq \frac{n \cdot (n-1)}{2}.$$



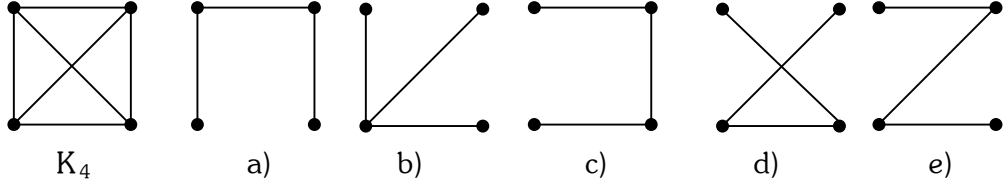
[Tic tac toe](#) oyunu... “Rakibin hamlesini nasıl bozarım?”
Okuyun, bu kavramlar daha iyi oturur, tahmin ediyorum.

Yayılım Ağaçları (Spanning Trees)

Önce derinliğine arama ya da önce genişliğine arama yordamları ile bitişik çizgede tüm düğümleri ziyaret ettiğimizde G çizgesindeki kenarları iki gruba ayırmış oluyoruz: (aslında 3'e 4'e ???)

- i) T: işlenilen kenarlar, ağaç kenarlar (*tree*)
- ii) N: işlenilmeyen, ağaç dışı kenarlar (*non-tree*) (*back-tree, cross-tree*)

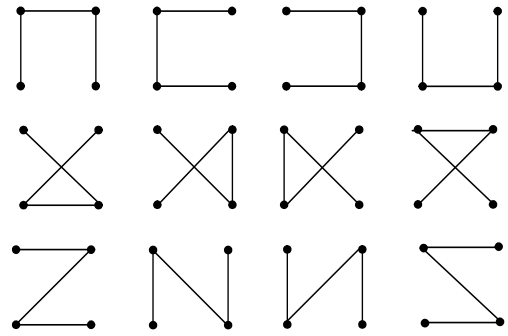
Tanım: Bir yayılım ağacı, tüm düğümlerin birbirine bitişik olmasını sağlayan olası kenarların oluşturduğu ağaçtır.



Eşbiçimlilik: K_4 çizgesi bir *complete graph*'tır. Üstte K_4 çizgesinden elde edilebilecek bazı yayılım ağaçları görülüyor. İlk ikisi (a, b) birbirinden farklı iki yayılım ağacıdır, fakat son üç tanesi (c, d, e) a'daki yayılım ağacı ile eşbiçimlidir (izomorfiktir). $a \equiv c \equiv d \equiv e$.

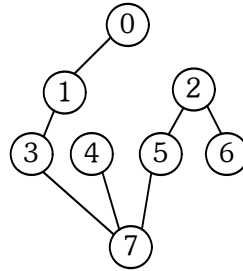
Kenar ağırlıkları: Şu an kenar ağırlıkları olmayan çizgelerle ilgileniyoruz. Eğer kenar ağırlıkları da devreye girerse "Kaç tane çizilebilir?" sorusunu yeniden ele almak gerekir.

Proje: Güzel bir 4. sınıf projesi olabilir: 3 boyutlu gösterimle $a \equiv e$ gösterimini yapan program, algoritma... Hangi düğümler yer değiştirmeli? Öncelikle izomorfizmin arka planındaki algoritmayı anlatmam lazım öğrenciye. O zaman iyi bir proje çıkarabilir.

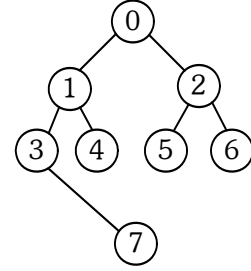


Bu yayılım ağaçlarının hepsi eşbiçimlidir.

Özel yayılım ağaçları: Önce derinliğine ve önce genişliğine aramada kalın gösterdiğimiz (dolaşılan) kenarlar birer ağaç oluşturmuştu. Önce derinliğine arama ile bulunan ağaca **önce derinliğine yayılım ağacı**, önce genişliğine arama ile bulduğumuz ağaca ise **önce genişliğine yayılım ağacı** diyoruz. Oluşan bu ağaçlara bir kenar daha eklense döngü (*cycle*) oluşur ve ağaçlar ağaç olmaktan çıkar.



Önce Derinliğine Yayılım Ağacı
(*depth-first spanning tree*)
(başlangıç 0 nolu düğüm)



Önce Genişliğine Yayılım Ağacı
(*breadth-first spanning tree*)
(başlangıç 0 nolu düğüm)

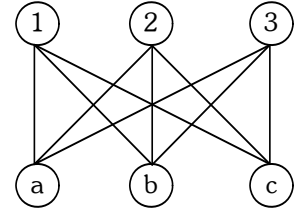
Soru: Kaç farklı yayılım ağacı vardır?

Yayılım ağaçlarında $n-1$ kenar bulunması gerekiyordu. Toplam e kenar içinden $n-1$ kenar $\binom{e}{n-1}$

farklı şekilde seçilebilir, bu kadar farklı durum vardır. Bütün bu durumları tek tek kontrol edip içlerinden döngü içermeyenleri tespit etmeliyiz. Ayrıca eşbiçimliler çıkabilir, bunları da hesaba katmalıyız.

Öncelikle bir tane yayılım ağacı elde edilir. Bu yayılım ağacına *non-tree* kenarlardan biri eklenir, böylece ağaçta döngü oluşur. Döngüyü oluşturan kenarlar tek tek çıkarılarak (yani çıkarılan geri eklenip başka bir tane çıkarılarak) her seferinde farklı bir yayılım ağacı elde edilir. Bu işleme sırayla tüm *non-tree* kenarlar kullanılana kadar devam edilir. Fakat bu algoritma uzundur, o yüzden bunu tercih etmek mantıklı değil.

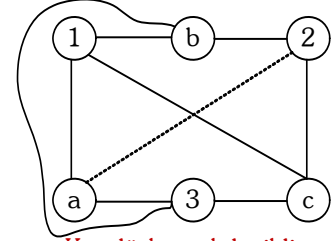
Şekilde $K_{3,3}$ çizgesinin eşbiçimli iki çizimi görülüyor. Bu ikisi aslında denk iki çizgedir. Eşbiçimliliği tespit edebilmek için önce derinliğine ve önce genişliğine aramadan yararlanılır. Bunlar zor problemlerdir. 4. sınıfta proje olarak isteyebilirim, ama iki dönem çalışmaları lazım. Çizgenin çizimini yapmak bile zor... Nasıl çizecekler? Kenarları olabildiğince kesiştirmeden ve kısa kenarlarla çizmelisin.



$K_{3,3}$ çizgesi

Kenarları kesişmeyen çizgeye düzlemsel (planar) çizge denir. Bu durumda $K_{3,3}$ çizgesi düzlemsel değildir. Çünkü $K_{3,3}$ çizgesini kenarların hiçbiri kesişmeksizin çizmenin yolu yoktur. Minimum tek bir kenar kesişerek çizilebilir.

Bu, düzlemselliği minimize edecek şekilde devre tasarlamaya yarar. Örneğin $K_{3,3}$ çizgesi şeklinde tasarlanan bir devreyi gerçekleştirmek için en az iki katman gereklidir. Bunlar zor problemler... Master ya da doktora yaparsanız, bunlarla uğraşabilirsiniz. Bu arada “Master ya da doktora yapmadıkça problem çözemezsiniz.” demiş oluyoruz.



$K_{3,3}$ düzlemsel değildir

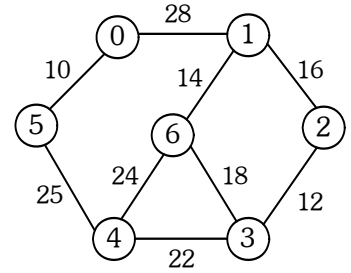
Minimum Yayılımlı Ağaçlar (Minimum Cost Spanning Trees)

Tanım: Yönsüz bir çizgede kenarlar ağırlıklandırılmış (*weighted*) olabilir. Çizgeyi $G(V,E)$ olarak tanımlıyorduk. Kenarlar ağırlıklı olduğunda ise $G(V,E,w)$ olarak tanımlarız. Yayılım ağaçları içerisinde kenar maliyeti en az olanla (ya da olanlarla) ilgileniyorsak minimum yayılımlı ağaç söz konusudur.

Örnekler:

a) Bir network sisteminde kenar maliyetleri bilgisayarlar arası bağlantı maliyetine karşılık gelsin.

b) 7 köye elektrik götürmek isteyebiliriz. Yapılan fizibilite çalışması sonrası bir köyden diğerine kaç km. fiber optik kablo kullanmak gerekeceğini tespit etmiş olabiliriz. Arasında bağlantı bulunmayan köyler arası sonsuz kabul edilir, onu yazmıyorum.



Algoritmalar: Minimum yayılım ağacı bulmak için çeşitli algoritmalar verilmiştir; Kruskal, Prim, Sollin. Biz bu 3 algortmadan ilk ikisini inceleyeceğiz. Konuyla ilgili diğer bir algoritma: reverse-delete algorithm

Backtracking, *greedy*, *dynamic* gibi programlama metotları vardır. *Greedy* geri dönmez, çözümünde ısrar eder. Bizim inceleyeceğimiz algoritmaların hepsi *greedy* (ısrarcı) algoritmalarlardır. *Greedy* yöntemi, her bir adımda optimal çözüm için en iyi kararın verildiği, ilerleyen aşamalarda bu kararın değiştirilmediği bir yaklaşımdır.

Minimum yayılımlı ağaçları bulurken şunlara dikkat ederiz:

- 1) Kenar seçerken döngüden kaçınıyoruz. Eğer bir kenar döngü oluşturuyorsa, o kenarı oluşturacağımız ağaçta kullanamayız. “Aşağı Tepecik Köyü’nden elektrik gelmezse Yukarı Tepecik’ten gelsin.” Yok!
- 2) En az maliyetliyi tercih ediyoruz.
- 3) n düğümlü çizgede $n-1$ kenarın seçilmesi yayılım ağacı oluşturmak için yeterlidir. Minimum yayılımlı ağaç da bir yayılımlı ağaç olduğu için $n-1$ kenar seçeceğiz.



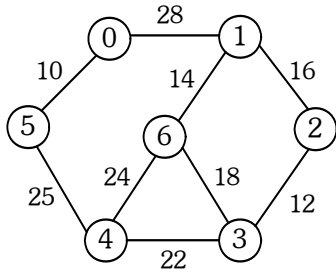
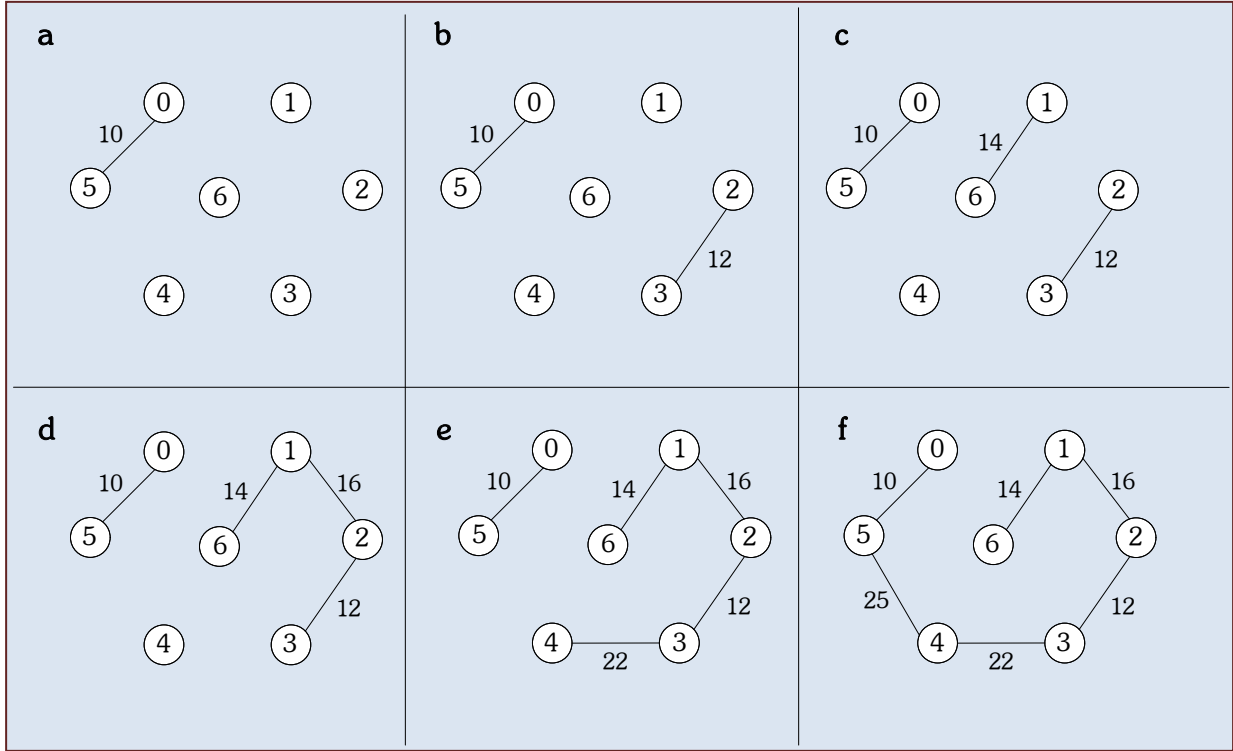
Kruskal Algoritması

1956'da *Joseph Bernard Kruskal* yazmış, literatüre **Kruskal algoritması** olarak girmiş. Kruskalın algoritması minimum yayılım ağacını döngüsüz olarak bulur; ispatı da var. [\[wiki\]](#)

T, seçilen kenarların kümesi; **E**, geri kalan kenarların kümesi; **|T|**, seçilen kenar sayısı olmak üzere Kruskal Algoritması aşağıda yarı algoritmik bir dille verilmiştir.

```
T = {};  
// boş bir seçilen kenarlar kümesiyle başla  
while( |T| < n-1 && E ≠ ∅ ) { // seçilmeyen kenarlar varken n-1 kenar seçene kadar  
    E kümesinden en az maliyetli bir (v,w) kenarı seç.  
    if( (v,w) kenarı T kümesi içinde bir döngü meydana getiriyor mu? )  
        getirmiyorsa (v,w) kenarını T kümesine ekle  
    else  
        (v,w) kenarını bırak  
}  
if( |T| < n-1 ) printf("minimum yayıllımlı ağaç bulunamadı.");
```

Burada **while(|T| < n-1 && E ≠ ∅)** döngü ifadesiyle ilgili birkaç şey açıklayalım. Yayılım ağacının n-1 kenardan oluştuğunu söylemiştik. Bu yüzden while döngüsü n-1 tane kenar seçene kadar çalışır. Eğer henüz n-1 kenar seçilememiş, fakat seçilebilecek hiçbir kenar da kalmamışsa döngü bu durumda da sonlanır. Doğru gibi gözüküyor, ama ispatını okuduğunuzda daha iyi anlarsınız.



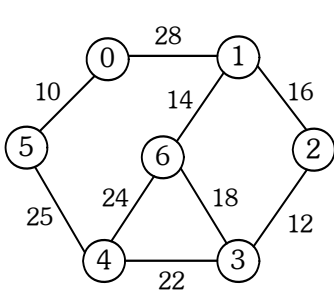
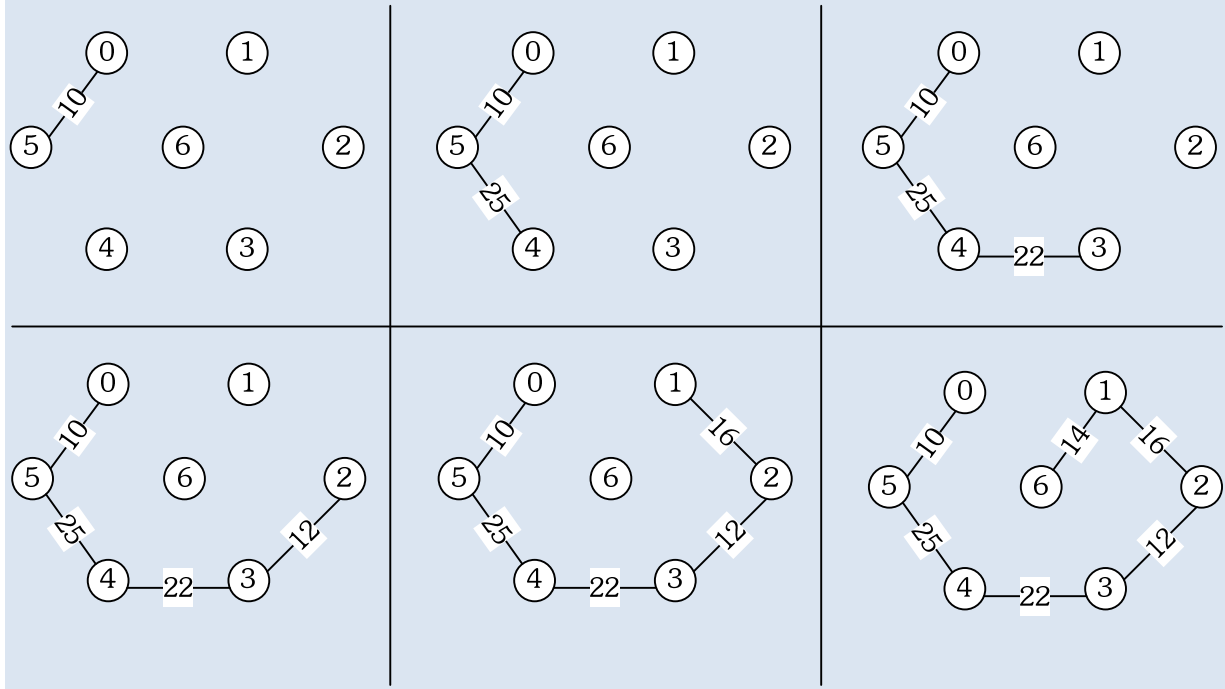
Kenar	Ağırlık	İşlem	Çözüm
-	-	Başlangıç	Tüm düğümler ayrık
(0,5)	10	Ağaca ekle	(a)
(2,3)	12	Ağaca ekle	(b)
(1,6)	14	Ağaca ekle	(c)
(1,2)	16	Ağaca ekle	(d)
(3,6)	18	Döngü var, ekleme yapma	
(3,4)	22	Ağaca ekle	(e)
(4,6)	24	Döngü var, ekleme yapma	
(4,5)	25	Ağaca ekle	(f)

Prim Algoritması

1930'da Vojtěch Jarník ve daha sonra 1957'de Robert C. Prim tarafından geliştirildi. 1959'da Edsger Dijkstra tarafından tekrar araştırıldı. Bu yüzden DJP algoritması, Jarník algoritması ya da Prim-Jarník algoritması olarak da bilinir. [\[wiki\]](#)

```
T = {};
TV = {0}; // 0 nolu düğümle başlıyoruz, TV (TreeVertex) düğüm kümesidir.
while( |T| < n-1 ){
    u ∈ TV ve v ∉ TV olacak biçimde en az maliyetli bir (u,v) kenarı seç.
    if( kriterlere uygun kenar yok ) break;
    TV = TV ∪ {v};
    T = T ∪ {u,v};
}
if( |T| < n-1 ) printf("minimum yayıllımlı ağaç yok.");
```

Kruskal'da ayrıntı kenarlar seçilebiliyor, en son adımda tek öbek haline geliyordu. Prim'in mantığı ise seçilen kenarlar kümesini tek öbek halinde büyütme. Bu yüzden dolayı her adımdaki '**seçilebilecek kenarlar**' kümesi, **seçilmiş düğümlerin seçilmemiş kenarlarından** oluşur. Yani seçilecek kenarın bir ucu seçilmiş düğümler kümesinde, diğer ucu ise seçilmemiş düğümler kümesinde olmalıdır. Bir ucu TV'de olacak, diğer ucu olmayacak. Tek bir öbek giderek büyüyor.



TV	T	Seçilebilecek Kenarlar
{0}	{}	{{(0,1), (0,5)}}
{0, 5}	{{(0,5)}}	{{(0,1), (5,4)}}
{0, 5, 4}	{{(0,5), (5,4)}}	{{(0,1), (4,3), (4,6)}}
{0, 5, 4, 3}	{{(0,5), (5,4), (4,3)}}	{{(0,1), (4,6), (3,6), (3,2)}}
{0, 5, 4, 3, 2}	{{(0,5), (5,4), (4,3), (3,2)}}	{{(0,1), (4,6), (3,6), (2,1)}}
{0, 5, 4, 3, 2, 1}	{{(0,5), (5,4), (4,3), (3,2), (2,1)}}	{{(0,1), (4,6), (3,6), (1,6)}}
{0, 5, 4, 3, 2, 1, 6}	{{(0,5), (5,4), (4,3), (3,2), (2,1), (1,6)}}	{{(0,1), (4,6), (3,6)}}

Toplam maliyetleri yazıyoruz; yani sona kadar toplam.

Hayatında 'algoritma' kelimesini duymamış çizge kuramcı bir adama problemi verirsin, o böyle çözer. Sen ise bunun arka plandaki veri yapısının ne olması gerektiğini keşfedersin. Problemi çizge kuramcı bir adam gibi çözebilecek seviyeye de gelmelisin.

Prim Ve Kruskal Algoritmalarını Kodlamak

Kruskal algoritmasının kodlamasını ödev olarak sorabilirim. Bakalım, veri yapılarını keşfedebilecek misiniz? Onu merak ediyorum. Bunları siz kodlayacaksınız:

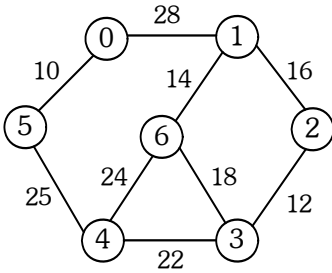
- 1) Döngü yapıyor mu?
- 2) En az maliyetli kenarı nasıl bulurum?

Bizim anlattıklarımız, çizge teorisindeki terminolojidir. “Biz de kodlarız.” dersiniz, veri yapılarını anlamamışsınızdır. Şimdi veri yapılarını anlamayan bir adam gibi kodlayacağım, biraz da veri yapılarını bilen bir adam gibi davranıp... Siz de iyi izleyin.

Kruskal Nasıl Kodlanmaz?

Algoritmanın çalışmasını gözlemlediğimizde önce 2 bileşen oldu, sonra 3 oldu. Daha sonra tekrar 2 bileşene düştü. Yakaladık! **Seçtiğim kenarları komşuluk listesinde tutarım**, oh çok güzel oldu.

Henüz **seçmediğim kenarları da bir dizide tutarım**. Kötü algoritma: Her seferinde en az maliyetliyi bulmak için arama yaparım, onu silip sonrakileri onun yerine doğru kaydırırım. Daha iyi algoritma: Kenarları kenar maliyetlerine göre sıralı olarak tutarım. (uç1, uç2, maliyet) **Sıralayarak çözersek $O(e)$ karmaşıklık.**



Şimdi çalıştıralım. (0,5) geldi, listede arattım. Ne 0 var, ne de 5... Her iki ucu da olmadığı için yeni bir listeye eklerim. Sonra (3,2), (6,1) geldi, bunlar da aynı şekilde yeni listelere eklenir. Sonra (1,2) geldi, 1 ve 2'yi arattım. Farklı listelerde bulunduklarından dolayı **döngü oluşmadı**. İki listeyi birleştiririm. Algoritma yavaş yavaş ortaya çıkıyor, ama **sürekli bir arama maliyeti** bulunduğunu gözlemlemeliyiz. (3,6) geldi, arattım. 3 de 6 da aynı listede, öyleyse **döngü oluşur**. Herkes anladı mı? Döngüyü tespit ettik, ama yanlış algoritmayı anlatıyorum. Böyle kodlamıyorsunuz, yoksa puan alamazsınız. “Böyle kodlamayın!” diyorum, ama doğru çalışıyor, onu anlatmaya çalışıyorum. Veri yapılarını alan kişi gibi şöyle kodlanır:

Kruskal Nasıl Kodlanır?

- 1) En az maliyetli kenarı seçmek için minimum öncelikli kuyruk veri yapısı kullanılır. Şimdilik kenarları tek tek *minheap*'e eklemeniz yeterli. Sıralama konusuna geçtiğimizde *heapsort* algoritmasını anlatacağız.
- 2) Döngü oluşturur mu? İki ayrı kümenin elemanıysa döngü olmaz. Anlattığımız “kümelerde union-find işlemleri” kullanılabilir. 3 ve 2 hangi kümeye ait? *Küme_bul()* yordamını anlattı mı hoca? Anlattı. İlkokulda öğretilen kümeler burada nasıl da işimize yarıyor.

Arka planda hoca hangi veri yapılarını anlatmış da haberimiz yokmuş!! Biz de gittik, liste tutuyoruz... Haftaya herkes Kruskal Algoritması'nı anlattığım şekilde kodlayıp getiriyor. Ben bunu anlatmasaydım en iyi kodlasanız, üstteki gibi kodlardınız. Fakat anlattığımız gibi kodlarsanız maliyet:

$$|E|.lg|E| + |V|.lg|E| + \Theta(1) = |E|.lg|E| + |V|.lg|E|$$

Buradaki $\Theta(1)$: Ters Ackermann fonksiyonu $\alpha(E,v) \leq lg|E|$ olduğu için böyle yazıldı. Eğer yığın oluştururken kenarları tek tek eklemek yerine elimizdeki diziyi yığın haline getirmek için *heapsort* kullanırsak karmaşıklık daha da düşer: $2.|E| + |V|.lg|E|$ olur.

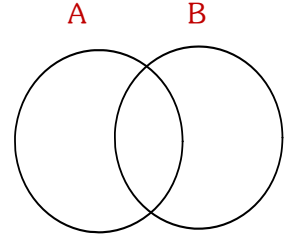
“Algoritmaya çok iyi hâkimiz.” Değil! Hala üzerinde çalışıyoruz. Problem çözerken neyi nerede kullanacaksın, işte veri yapıları burada devreye giriyor. Adam hayatı boyunca bunu öyle çözer, güzel de çalışır algoritma... Bu adama ne diyeyim ben şimdi?

Prim'i Kodlamak

Bunu sizin kodlamanız biraz zor olabilir. Anlatsam bile kodlayamayabilirsiniz.

Ben Kruskal'ı anlatırken hiç sıralama anlatmadım. Şimdi Prim'e geelim; bundaki zorluk:

Algoritma, en az maliyetli kenarı tek hamlede alacak. Bir kenar seçildiğinde sonraki adımda seçilebilecek kenarlar kümesine yeni kenarlar eklenebilir ve eski kenarlardan bazıları çıkabilir. '*Seçilebilecek kenarlar*' sorgulamasında öyle bir şey yapmalısın ki, her aşamada '*seçilebilecek kenarlar*' kümesi hemen getirilmeli. **Bir sonraki aşamada seçilebilecek kenarlar kümesi anında değişmeli! A kümesi iken B oldu.** *Advanced Data Structures* anlatıyorum. Örneğin 0 düğümü seçildiğinde sadece (0,1) ve (0,5)'i gören bir veri yapısı olmalı. Sadece bu kenarlardan minimum maliyetlisini seç, diğerlerine bakma bile!



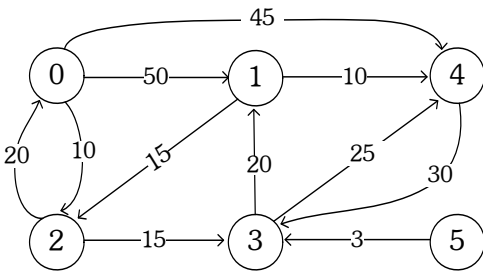
Bunun çözümünü bulabiliyorsanız, veri yapıları sizi bir noktaya taşımıştır. Bunu yapmak önemli... **MIT, Algorithms** kitabında bulabilirsiniz, var diye hatırlıyorum. "Zaten Prim yazmış, ben de kodlarım." dersin karmaşıklık birden büyür.

Problem Çözmede Yaklaşım: Bu problem nasıl çözülmüş? Literatürde hangi veri yapıları önerilmiş? Dünya bunu nasıl çözüyor, ben nasıl çözeyim? Ben de yapıyorum kenarda tıkr tıkr bir şeyler...

"To follow the path:
look to the master, follow the master,
walk with the master, see through the master,
become the master."

Araştırma: Kruskal ve Prim algoritmalarından hangisi, hangi tür çizgelerde daha etkilidir?

En Kısa Yol Problemi (Shortest Path)



Şekildeki çizgede düğümlerin şehirleri, kenarların ise iki şehir arasındaki demir yolu uzaklığını gösterdiği farz edelim. Bir şehirden diğerine gitmek için en kısa yol hangisidir?

Şekildeki çizgede kenarlar bir ağ ortamında iki düğüm arasındaki trafik sıkışıklığını ifade etsin. Yapılan istatistikler sonucunda her hangi iki bilgisayar arası paket gönderiminin ne kadar geciktiği ayrı ayrı tespit edilmiştir. Bu verilere göre bir *router* nasıl yazılmalıdır? [\[wiki\]](#)

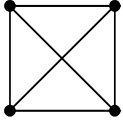
Kaynak	Hedef	Maliyet
0	5	Yol yok!
0	2	10
0	3	25
0	1	45
0	4	45

En kısa yol problemi;

- 1) x nolu düğümden y nolu düğüme yol var mı?
- 2) Eğer x düğümünden y düğüme birden fazla yol varsa en kısa olanı hangisidir?

Tüm düğümlere bir kez uğrayarak başlangıç noktasına dönmek, **gezgin-satıcı problemi**, daha farklı ve zor bir konudur.

Tek Kaynaktan Tüm Hedeflere En Kısa Yol (Single Source All Destinations)

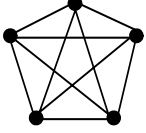


K₄

K₄ çizgesi, tüm düğüm ikilileri arasında kenar bulunduğundan dolayı bir *complete graph*'tır, ayrıca düzlemseldir. Eğer **bir kaynaktan bir hedefe** en kısa yolu **brute force** yöntemiyle bulmaya çalışırsak olası yollar (kaynak=0, hedef = 3):

n=4 için toplam **5 yol**:

0-3	0-1-3	0-2-3	0-2-1-3	0-1-2-3
-----	-------	-------	---------	---------



K₅

K₅ çizgesi, hiçbir kenarı kesismeksizin çizilmesi mümkün olmadığından dolayı düzlemsel (*planar*) değildir. (kaynak=0, hedef = 4): n=5 için toplam 1+3+6+6 = **16 yol**:

0-4	0-1-4	0-2-4	0-3-4	0-1-2-4	0-1-3-4	0-2-1-4	0-2-3-4
0-3-1-4	0-3-2-4	0-1-2-3-4	0-1-3-2-4	0-2-1-3-4	0-2-3-1-4	0-3-1-2-4	0-3-2-1-4

Brute force'un analizi: Olası yolların sayısını veren genel bir formül aşağıdaki biçimde modellenebilir. Bu modelleme kaba bir analizdir, ayrıntılı değildir. Modelleme doğrudur, sonuç yuvarlak verilmiştir.

$$1 + \binom{n-2}{1} + \binom{n-2}{1} \cdot \binom{n-3}{1} + \binom{n-2}{1} \cdot \binom{n-3}{1} \cdot \binom{n-4}{1} + \dots + \binom{n-2}{1} \cdot \binom{(n-2)-1}{1} \cdot \dots \cdot \binom{(n-2)-(n-3)}{1}$$

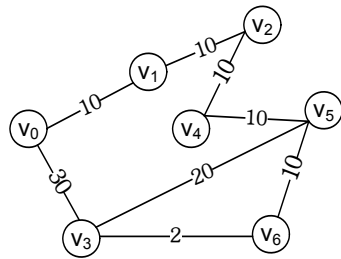
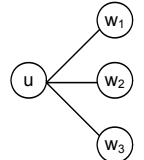
$$= 1 + (n-2) + (n-2)(n-3) + (n-2)(n-3)(n-4) + \dots + (n-2)! \approx 1 + O(n) + O(n^2) + O(n^3) + \dots + O(n^n) \approx O(n^n)$$

Görüldüğü gibi *brute force* yöntemiyle tek bir kaynaktan tek bir hedefe en kısa yolu bulurken bile karmaşıklık $O(n^n)$ oldu. **Tek bir hedef için bile böyle...** Elbette mantık doğru, ama böyle çözmeyeceğiz. Hatırlayın, *brute force* ile 5x5'lik küçük bir matrisi doldurmak için bile asırlar gerektiğini görmüştük.

Şimdi etkin çözüme doğru ilerliyoruz: En kısa yolları bulmak için yine *greedy* mantığını kullanacağız. S kümesi en kısa yolun düğümler kümesi olsun. (0 nolu başlangıç düğümünü de içerir.) S kümesinde olmayan bir w düğümü için $uzaklık[w]$; 0 nolu düğümden başlayan ve S kümesinde yer alan bazı düğümlerden geçerek w düğümünde son bulan en kısa yolun uzunluğunu tutsun.

$uzaklık[u]$ değeri minimum olan bir u düğümü seçilir ve S kümesine eklenir. u düğümü seçildiğinde, S kümesinde olmayan düğümlerin $uzaklık[]$ 'ları için tekrar en kısa yol aranmalıdır. Çünkü u düğümünün S kümesine eklenmesi, S 'de bulunan düğümlerden geçip S 'de bulunmayan herhangi bir w düğümünde ($w \notin S$) sonlanan yolların en kısa $uzaklık[]$ 'ını değiştirebilir.

Örneğin; u düğümünün eklenmesiyle u 'nın komşusu olan w_1, w_2, w_3 düğümlerine **u üzerinden geçilerek gidilen yeni yollar** açılmış olacaktır. Bu yeni yolların uzunluğu, o ana kadar bulunmuş olan eski yollardan daha kısa olabilir. Eğer bu durum gerçekleşirse w 'ya geliş maliyeti azalmış olur, **u 'dan w 'ya daha kısa bir yol bulunmuştur**, $uzaklık[w]$ değeri güncellenmelidir. $u-w$ alt yolunda ara düğüm bulunmadığı için $0-w$ yolunun uzunluğu $uzaklık[u] + maliyet(<u, w>)$ olur.



Şekildeki çizge üzerinde v_0 düğümünden tüm düğümlere en kısa yolu bulmaya çalışalım. Gidilebilecek tüm düğümleri $v_1, v_2, v_3, v_4, v_5, v_6$ sırasıyla S kümesine ekleyeceğiz.

gidildi	uzaklık	gidildi	uzaklık	gidildi	uzaklık	gidildi	uzaklık	gidildi	uzaklık	gidildi	uzaklık	gidildi	uzaklık
0 T	0	0 T	0	0 T	0	0 T	0	0 T	0	0 T	0	0 T	0
1 F	10	1 T	10	1 T	10	1 T	10	1 T	10	1 T	10	1 T	10
2 F	∞	2 F	20	2 T	20	2 T	20	2 T	20	2 T	20	2 T	20
3 F	30	3 F	30	3 F	30	3 T	30	3 T	30	3 T	30	3 T	30
4 F	∞	4 F	∞	4 F	40	4 F	40	4 T	40	4 T	40	4 T	40
5 F	∞	5 F	∞	5 F	50	5 F	50	5 F	50	5 T	50	5 T	50
6 F	∞	6 F	∞	6 F	60	6 F	60	6 F	60	6 F	60	6 T	60

$S = \{v_0\}$ ile başladık. v_4 düğümü eklendiğinde v_5 'e olan uzaklık azaldı, yukarıda anlatılan durum oluştu. Demek ki, farklı bir yoldan geliş, daha kısa olabiliyor. İşte, bir düğümün eklenmesi başka düğümlere gidiş maliyetini bu şekilde değiştirebilir.

Not: Arada bağlantı yoksa (sonsuz denmesi gerekiyorsa) çok büyük bir sayı (1000) ile bunu ifade ettik.

```
1: #define MAX_DUGUM 6
2:
3: int maliyet[][MAX_DUGUM] = {
4:     { 0, 50, 10, 1000, 45, 1000},
5:     {1000, 0, 15, 1000, 10, 1000},
6:     { 20, 1000, 0, 15, 1000, 1000},
7:     {1000, 20, 1000, 0, 35, 1000},
8:     {1000, 1000, 100, 3, 1000, 0}};
9: int uzaklik[MAX_DUGUM];
10: short int gidildi[MAX_DUGUM];
11: int n = MAX_DUGUM, v=0;
12:
13: void EnKisaYolBul( int v, int maliyet[][MAX_DUGUM]
14:                  int uzaklik[], int n, short int gidildi[] ){
15:     int i, u, w;
16:     int minimumBul( int*, int, short int * );
17:
18: A1: for( i=0 ; i<n ; i++ ){
19:     gidildi[i] = FALSE;
20:     uzaklik[i] = maliyet[v][i];
21: }
22: gidildi[v] = TRUE;
23: uzaklik[v] = 0;
24:
25: A2: for( i = 2 ; i < n ; i++ ){ // n-1 adet yol bul
26: A3:     u = minimumBul( uzaklik, n, gidildi );
27:     gidildi[u] = TRUE;
28:
29: A4:     for( w = 0; w < n ; w++ )
30:         if( !gidildi[w] )
31:             if( uzaklik[u] + maliyet[u][w] < uzaklik[w] )
32:                 uzaklik[w] = uzaklik[u] + maliyet[u][w]; //maliyet değişti
33:     }
34: }
35:
36: int minimumBul( int uzaklik[], int n, short int gidildi[] ){
37:     int i, min, pos;
38:     min = INT_MAX;
39:     pos = -1;
40:     for( i=0 ; i<n ; i++ )
41:         if( uzaklik[i] < min && !gidildi[i] ){
42:             min = uzaklik[i];
43:             pos = i;
44:         }
45:     return pos;
46: }
```

Analiz: Algoritma karmaşıklığı $n + n[n + n] = n + 2n^2 = O(n^2)$ olur. Buradaki n'ler sırasıyla A1, A2, A3 ve A4 etiketli satırlardan geliyor. Adam matristeki en büyük elemanı bulana kadar, ben v_0 'dan tüm düğümlere olan en kısa yolları buluyorum.

Yapıları iyileştirmek: Daha iyi bir algoritma yazabilirsin. Uzaklıkları tutmak için minimum öncelikli kuyruk kullanırsak 3 numaralı n yerine $\log_2 n$ gelir. Demek ki *minheap*'i burada da kullanabilirmişiz. Minimum öncelikli kuyruğu öğrendikten sonra dizide en küçük elemanı aramak ne kadar hantal kalıyor değil mi? Ne kadar abes! Ne kadar ayıp! Düğümleri komşuluk listesinde tutabiliriz. Bu durumda 4 numaralı n yerine $\text{degree}(v_i)$ gelecektir. Son durumda karmaşıklık $n + n[\max(\lg n + \text{degree}(v_i))]$ = $O(n \cdot \lg n)$ olur. Bakın, 3 farklı algoritma anlattım; Brute force: $O(n^2 n^n)$, kitaptaki algoritma: $O(n^2)$, yapılar iyileştirilince: $O(n \cdot \lg n)$.

Yolu bulmak: Şimdi maliyeti bulduk da yolu nasıl bulacağız? Maliyeti ben buldum, yolu da sen bul! Bu program üzerinde gerekli veri yapısı değişikliğini yaparak maliyetle birlikte yolu da söylemesini sağlayınız. Hangi yoldan gidileceği belli olmadığından dolayı *forward* çözemezsiniz. **Tek bir diziyle çözülebilir: önceller dizisi. Problem çözüldü; peki, nasıl kullanacağız? Tersten kullanacaksın.**

Yazılım Mühendisliği Kavramları

Vizedeki soruyu tek bir özyineli yordamla çözmeye çalışmışsınız, birkaç kişi gerçekten çözmüş. Ama bu çözüm hem modüler değil, hem de okunabilirliği az. Orda bir hile yapıyorsunuz. Bazıları da yordamı `AgirlikBul(...)`; diye çağırmış. Bu ne? “*Hocam ne var ki orda?*” Bu ne döner? “*int döner.*” Hani nereye dönüyor? Seninki havada duruyor. “*Ah, ben onu unutmuşum.*”

Çözümlerinizi modüler, basit, okunabilir, tekrar kullanılabilir, bakımı kolay... olmalıdır. Yazılım mühendisliği kavramlarının çıktısını alıp kodlarınızı bu özelliklere uygun üretin. Elinizin altında tıpkı *Assembly vs. kitabı* bulunduğu gibi bir de Yazılım Mühendisliği kitabı bulunmalı. Bu kavramlardan bazıları: [[Tips to succeed in Software Engineering Student Projects](#)]

Reliability
Portability
Testability
Understandability
Modifiability

Usability
Integrity
Efficiency
Modularity
Correctness

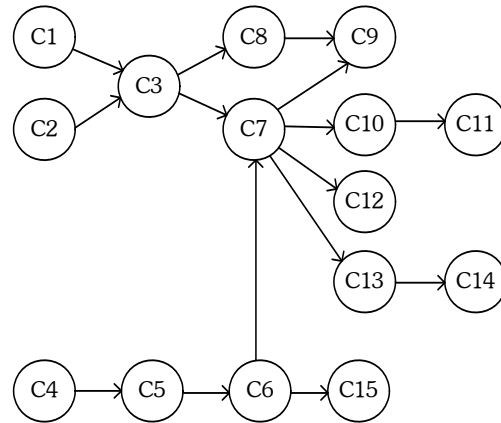
Maintainability
Flexibility
Interoperability
Traceability
Expandibility

Aktivite Ağları

Genellikle projeler, **aktivite** adı verilen alt projelerden oluşur. Proje, tüm aktiviteler tamamlandığında bitirilmiş olacaktır. Eğer **proje**, bir üniversiteden mezun olmak ve ilgili dereceyi almak ise, **aktiviteler** (faaliyetler) derslere karşılık gelecektir.

Aktiviteler, inşaat mühendisliğinde daha çok geçer. İlk katın temeli atılır, 21 gün sonra 2. katın temeli atılır. Kabası bitince başka bir işe geçilir. Bazı işler arasında **bağlantılar** var, birbirinden **bağımsız işler** de var. Proje ne kadar zamanda **biter**? Hangi işler **kritiktir**, yani aksarsa projenin bitme süresi uzar? Hangi işlerde **bolluk** vardır, aksasa da projenin bitme süresi değişmez?

Ders kodu	Ders adı	Ön gereksinim
C1	Programming I	-
C2	Discrete Math.	-
C3	Data Structures	C1, C2
C4	Calculus I	-
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Lang.	C3
C9	Operating Systems	C7, C8
C10	Programming Langs.	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	-



Derslerin kimisi bağımsız bir biçimde alınabilir. Kimi dersler ise bazı derslerin alınması için bir ön şart olabilir, aralarında bir **öncül ilişkisi** (*precedence relation*) bulunabilir. **Öncül ilişkisini yönlü bir çizge ile gösterebiliriz.** Eğer *i* dersi *j* dersinin öncülü ise çizgemizde bunu ifade eden **<i,j>** kenarı bulunacaktır.

Satır arası: Yöneylem, sayısal analiz gibi derslerin kalkmış olması, sizin, matematiksel olarak zayıf kalmanıza sebep oluyor.

Tanım: Düğüm Aktif Ağlar (activity on vertex / node networks – AoV / AoN)

Ağda **aktiviteler düğümlerle**, öncül ilişkileri ise kenarlarla temsil edilir. C3 ve C6 dersleri, C7 dersinin **yakın öncülleridir** (*immediate predecessors*). C9, C10, C12, C13 dersleri C7 dersinin **yakın ardıllarıdır** (*immediate successor*). C14 bir ardıl olmakla birlikte, C3'ün yakın ardılı değildir. **Düğüm aktif** kavramını master öğrencileri çalıştı, kavram artık oturdu.

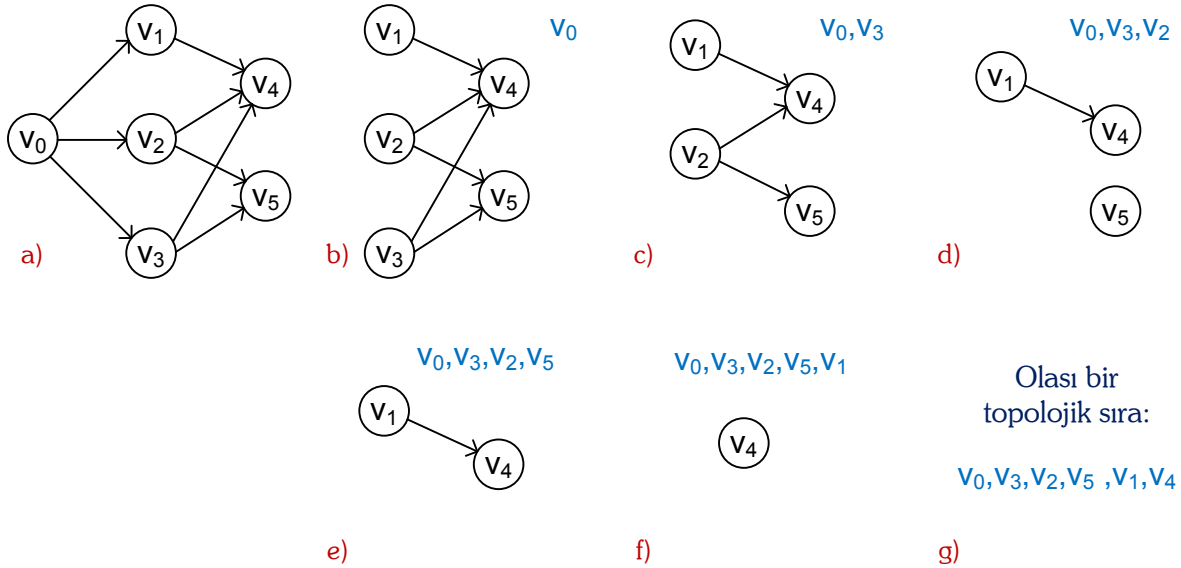
Tanım: Topolojik Sıra (topological ordering / topological sort)

İkinci dünya savaşı sırasında, füze üretimini daha hızlı yapabilmek için keşfedilmiştir. **Topolojik sıra**, düğümlerin, yakın-ardıl yakın-öncül ilişkileri dikkate alınarak verilen doğrusal bir sırasıdır. Her hangi bir düğüm ikilisi (i, j) verildiğinde eğer i, j 'nin öncülü ise, söz konusu doğrusal sıra içerisinde i, j 'den önce yer alır. Topolojik sıra **birden çok olabilir**. Örneğin bilgisayar bilimleri derecesini hak etmek için alınması gereken derslere ilişkin topolojik iki farklı topolojik sıra verilmiştir; sağlama yaparak öncül ilişkisine uyulduğunu görebilirsiniz: [\[wiki\]](#)

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

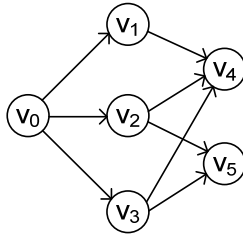
C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

Topolojik sıra bulmak: Bir çizgeye ilişkin topolojik sıranın nasıl bulunacağını aşağıdaki örnek ile verelim.

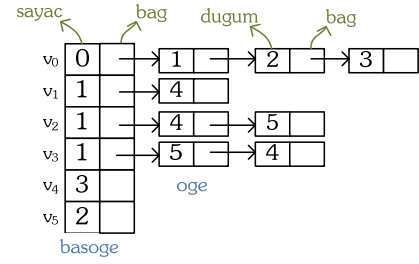


```
1: typedef struct oge *oge_gost;
2: typedef struct oge {
3:     int dugum;
4:     oge_gost bag;
5: };
6:
7: typedef struct {
8:     int sayac; // <i,j> kenarı verildiğinde baş düğümü j olan kenar sayısı
9:     oge_gost bag;
10: } basoge;
11:
12: basoge baslik[MAX_DUGUM];
```

Topolojik Sıra Bulma Yordamı



Olası bir
topolojik sıra:
0, 3, 2, 5, 1, 4



```
1: void TopolojikSira( basoge baslik[], int n ){
2:     int i, j, ust = -1;
3:     oge_gost p;
4:     for( i=0 ; i<n ; i++ )           // her döngüde bir düğüm basılacak
5:         if( !baslik[i].sayac ){       // öncülü olmayan düğümleri
6:             baslik[i].sayac = ust;    // yığıta koy
7:             ust = i;
8:     for( i=0 ; i<n ; i++ )
9:         if( ust == -1 ){ printf("\nYonlu cizgede dongu var."); break; }
10:    else {
11:        j = ust;                       // yığıttan bir düğüm al
12:        ust = baslik[ust].sayac;       // alınan düğümü yığıttan kaldır
13:        printf("%d ", j);             // j düğümü işlendi
14:        for( p = baslik[j].bag ; p ; p = p->bag ){
15:            int k = p->dugum;          // k düğümü j düğümünün ardılı
16:            baslik[k].sayac--;         // k'nın öncül sayısını azalt
17:            if( !baslik[k].sayac ){    // k'nın öncül sayısı 0'a düştü mü
18:                baslik[k].sayac = ust; // k'yı yığıta koy
19:                ust = k; } } }
```

Ortada yığıt falan yok, ama [6:] “yığıta ekle”, [11:] “yığıttan al” diye açıklama satırları yazdık. Açıklama satırları, işte bu yüzden dolayı önemlidir. Yığıt kullanıyor da, nasıl kullanıyor ya?

`baslik[i].sayac`, i düğümüne kaç düğümden geldiğini, yani i düğümünün öncüllerinin sayısını tutuyordu. Eğer bu değer 0 ise bu düğümün öncülü yoktur, topolojik sıraya yazılabilir. Aynı anda `sayac` alanı 0 olan birden çok eleman da bulunabilir. Öyleyse bunların birini yazdırırken diğerlerini de unutmamalıyız.

[6-7:] Öncül sayısı 0 olan düğümleri yığıta koyuyoruz, buradaki anlamıyla peş peşe bağliyoruz. Kümelerde kullandığımız ata dizisine benziyor. İşte geriye doğru ilerlemeyi sağlayan bu küme benzeri yapı, bir anlamıyla yığıttır. Yığıt kavramını bu şekilde de gerçekleştirme imkânımız var.

[11:] Topolojik sıraya yazmak üzere yığıttan 1 tane eleman alınız ve [13:] yazarsınız. Artık bu düğüm işlenmiş oldu. [16:] Bir iş bittiğinde, başlaması bu işe bağlı olan işlerin öncül sayısı 1 azalmış olur. [17:] Bu azaltma sonrası öncül sayısı 0'a düşen düğüm olursa, o da topolojik sıraya yazılabilecek hale gelmiştir, [18:] yığıta eklenmelidir.

[10:] Bu kod sadece topolojik sırayı bulmaz, varsa döngüyü de bulur. Döngü demek, A ve B düğümleri karşılıklı olarak birbirlerinin öncülü olması demektir ki, bu bir hatadır. Çünkü döngü varsa proje hiç bitmeyecektir.

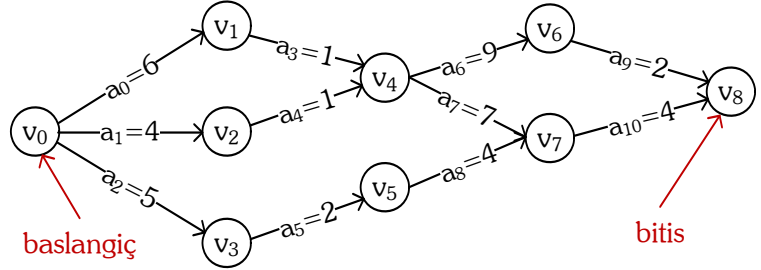
Analiz: İlk `for` $O(n)$, ikinci `for` $O(n)$, `else` bloğu d_i (i 'nin çıkan derecesi) olduğundan karmaşıklık;

$n + n \cdot [\max(\text{out-degree}(v_i))] = O\left(\sum_{i=0}^{n-1} d_i + n\right) = O(e + n)$ olur. Bu da çizgedeki düğümlerin birbirine bağlanışına göre $O(n)$ ile $O(n^2)$ arasında değişebilir.

				sayac alanları						Çıktı
ust	i	j	k	v ₀	v ₁	v ₂	v ₃	v ₄	v ₅	
-1	?	?	?	0	1	1	1	3	2	
0	?	?	?	-1	1	1	1	3	2	
-1	0	0	1	-1	0	1	1	3	2	0
1	0	0	2	-1	-1	0	1	3	2	
2	0	0	3	-1	-1	1	0	3	2	
3	0	0	3	-1	-1	1	2	3	2	
2	1	3	5	-1	-1	1	2	3	1	3
2	1	3	4	-1	-1	1	2	2	1	
1	2	2	4	-1	-1	1	2	1	1	2
1	2	2	5	-1	-1	1	2	1	0	
5	2	2	5	-1	-1	1	2	1	1	
1	3	5	5	-1	-1	1	2	1	1	5
-1	4	1	4	-1	-1	1	2	0	1	1
4	4	1	4	-1	-1	1	2	-1	1	
-1	5	4	4	-1	-1	1	2	-1	1	4

Tanım: Kenar Aktif Ağlar (activity on edge / arc networks – AoE / AoA)

Kenar aktif ağlar, düğüm aktif ağlarla yakından ilişkilidir. Düğüm-aktif ağlardakinin tersine çizgedeki **yönlü kenarlar faaliyetleri** (eylem-aktivite) temsil eder. Öncel ilişkileri ise düğümlerle gösterilir. Varsayımsal bir projenin çizgesi aşağıda verilmiştir. **Örneğin;** a_{10} aktivitesinin başlayabilmesi için a_7 ve a_8 aktiviteleri bitmelidir. Görüldüğü gibi bitmesi gereken iş, bir düğümle ifade ediliyor.



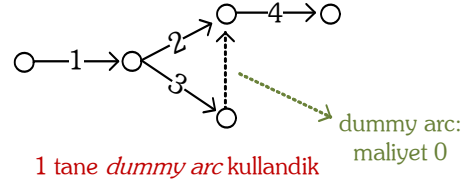
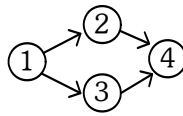
Graph Drawing

Yukarıda kenar-aktif çizgesi verilen örnek projede 11 adet aktivite, 9 adet olay (*event*) bulunuyor.

Olay: Bir projenin tamamlanması için geçen sürede ulaşılması gereken ara aşamalardır.

Yukarıdaki projede a_0, a_1, a_2 aktiviteleri paralel olarak başlayabilir. a_0 'dan sonra a_3, a_1 'dan sonra a_4, a_2 'dan sonra a_5 aktivitesi başlar. a_3 ve a_4 aktiviteleri tamamlandıktan sonra a_6 ve a_7 aktiviteleri başlar. a_{10} aktivitesinin başlayabilmesi için a_7 ve a_8 aktiviteleri tamamlanmış olmalıdır. v_0 projenin **başlangıç**, v_8 ise **bitiş** noktalarıdır. [wiki] [graphdrawing.org] [dmoz.org]

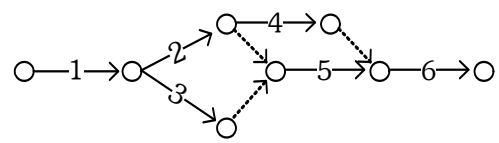
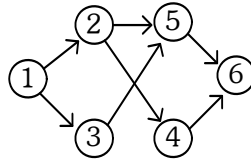
Düğüm-aktif çizmeyi kenar-aktif çizmeye çevirmek: Her düğüm-aktif çizge, bir kenar-aktif çizge ile gösterilebilir. Fakat düğüm aktif çizmeyi kenar aktif çizmeye çevirmede bazen zorluk yaşanabilir. Gösterimde yaşanan bu zorluklar, yapay aktiviteler eklenerek çözülür.



1 tane *dummy arc* kullandık

dummy arc: maliyet 0

Yapay aktivite (*dummy activity*): Maliyeti 0 olan kenarlardır. = Yapay kenar (*dummy arc*) Örneğin yukarıdaki projede eğer a_6 ve a_7 aktiviteleri; a_3, a_4 ve a_5 aktiviteleri tamamlandıktan sonra başlamalı koşulu bulunsaydı, ağ yapısına $\langle v_5, v_4 \rangle$ yapay aktivitesi (*dummy activity*) eklemek gerekecekti.



3 tane *dummy arc* kullandık

Minimum arc problem: Çizmeyi, en az sayıda yapay aktivite kullanarak çizibilme problemi. Bu konuda dünyada birkaç master tezi, bir tane de doktora tezi var. 3 arkadaşın tezi, oldukça ilerledi. *NP-Complete (nondeterministic polynomial time)* karmaşıklık düzeyinde bir problemdir.

Bu konuyu ele alan çözümleme yöntemleri: [wiki]

PERT (*Program/Project Evolution and Review Technique*) 1958

CPM (*Critical Path Method*) 1950

RAMPS (*Resource Allocation and Multi project Scheduling*)

En Erken ve En Geç Başlama Zamanı (*Earliest Times*)

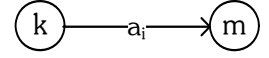
Projenin tamamlanmasının ne kadar süreceğini bilmek isteriz. Üzerinde bulunan aktivitelerin süreleri toplandığında maksimum değeri veren v_0-v_8 yolu (*path*), **kritik yol** (*critical path*) olarak isimlendirilir. Birden fazla kritik yol bulunabilir. Yukarıdaki projede v_0, v_1, v_4, v_7, v_8 yolu da; v_0, v_1, v_4, v_6, v_8 yolu da birer kritik yoldur. Bu durumda söz konusu **projenin en erken tamamlanma süresi 18** olarak bulunur.

Belli bir aktivite, ön koşulu olan tüm aktiviteler olabildiğince kısa sürede bitirilirse, ne zaman başlar? Ya da projenin tamamlanma süresinin uzamaması için, belli bir aktiviteye en geç ne zaman başlamak gerekir? Hangi işlerde aksama olursa proje süresi uzamaz? Hangileri aksarsa proje de aksar? Bunları bilmeliyiz ki, kritik işlerin ustalarına “Sakin 4 günde bitirin, 5 güne sarkmayın. 3 günde biterse memnun olurum.” diyebilelim. Bu, yön-eylem problemidir.

2. Vize Sorusu: 2. vizede en erken ve en geç başlama zamanı hesaplamayı sordum, yine sorarım. Bunu veririm, “İşletin.” derim. Anladım zannedersiniz, ama %80'i işlemez.

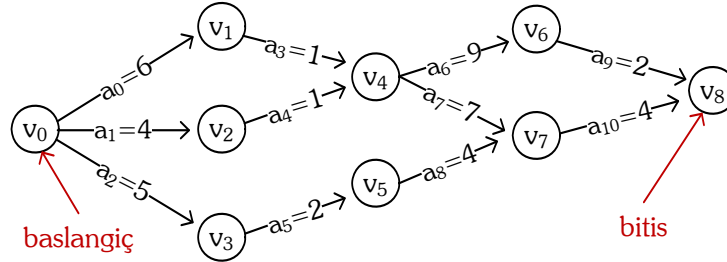
En Erken Başlama Zamanlarının Hesaplanması (Calculation of Earliest Times)

Bir a_i aktivitesinin en erken başlama zamanı $EnErken(i)$, en geç başlama zamanı $EnGec(i)$ ile sembolize edilsin ve bir a_i aktivitesi $\langle k, m \rangle$ kenarı ile temsil edilsin.



a_i 'nin en erken başlama zamanı demek, k düğümünün en erken başlama zamanı demektir:

$$EnErken(i) = EnErkenBas[k]$$



Örneğin a_1 aktivitesinin en erken başlama zamanı $k=v_0$ olduğundan 0; a_5 aktivitesinin en erken başlama zamanı ise $k=v_3$ olduğundan 5 (a_2); a_8 aktivitesinin en erken başlama zamanı ise $k=v_5$ olduğundan $5+2=7$ 'dir ($a_2 + a_5$). Öyleyse düğümler için $EnErkenBas[]$ dizisi tutacağız.

Başlangıç düğümü (v_0 olayı) için $EnErkenBas[0] = 0$ kabul ettik. Çünkü **projeye başlama zamanını 0 kabul ediyoruz**. Gerçekte bununla, “8 Mayıs 2009” gibi bir başlama tarihini ifade etmiş oluyoruz.

Yukarıdaki v_3 ve v_5 ile ilgili açıklamada olduğu gibi v_0 dışındaki bir v_j düğümü için en erken başlama zamanını v_0 'dan v_j 'ye gelen yoldaki aktivite sürelerini toplayarak buluruz. Fakat **her zaman tek yol olmayabilir**. Örneğin v_4 düğümüne gelen iki yol vardır. **Bu durumda hangi yolu tercih edeceğiz?**

v_4 düğümünden başlayan a_6 ve a_7 aktivitelerinin başlayabilmesi için v_4 düğümünde sonlanan a_3 ve a_4 aktivitelerinin her ikisinin de bitmiş olması gerekir. a_3 aktivitesi en erken $6+1=7$. zamanda tamamlanabilir. a_4 aktivitesi ise en erken $4+1=5$. zamanda tamamlanabilir. Her iki aktivitenin de bitmesi için kısa işler bitse bile en uzun işin, yani 7. zamanın beklenmesi gerektiği açıktır. Demek ki, bir düğüm için alternatif yollar varsa, **bu yolların en uzun olanı, düğümün en erken başlama zamanını verir**. $P(m)$: m düğümünün yakın öncüllerinin oluşturduğu küme olmak üzere $k \in P(m)$ ise;

$$EnErkenBas[m] = \max \{ EnErkenBas[k] + \langle k, m \rangle \text{ aktivitesinin süresi} \}$$

Öyleyse en erken başlama zamanını araştırırken, o ana kadar bulunmuş olandan daha uzun bir yol bulunursa $EnErkenBas[k]$ verisi güncellenmelidir. Örneğin; v_4 için 5 uzunluğundaki yola daha önce rastlanmışsa $EnErkenBas[4]=5$ yazılmıştır. Sonra daha uzun bir yol olan ($7>5$) $v_0-v_1-v_4$ yolu bulunduğu zaman $EnErkenBas[4]=7$ olarak güncellenmelidir:

```
if( EnErkenBas[m] < EnErkenBas[k] + p->süre )
    EnErkenBas[m] = EnErkenBas[k] + p->süre;
```

Bu noktaları açıkladıktan sonra belirtilmesi gereken son bir şey daha var. Aradan bir düğüm seçip en erken başlama zamanını bulmak mümkün müdür? Hayır, değildir. **Çözümleme, topolojik sıraya göre yapılmalıdır**. Zaten açıklanan çözümleme yöntemi de bunu gerektirmektedir; çünkü bir düğümün en erken başlama zamanını bulmak için öncüllerinin en erken başlama zamanlarını bilmemiz gerekmektedir.

Daha önceden yazdığımız $TopolojikSira()$ yordamında yapılacak küçük bir değişiklikle en erken başlama zamanını bulan yordamı yazmış olacağız. Kullanılan yapıya $süre$ alanını ekleyip sonra da $TopolojikSira()$ 'da sayac alanını azaltan 16. satırdan önceye şu 2 satır kodu eklememiz yeterlidir:

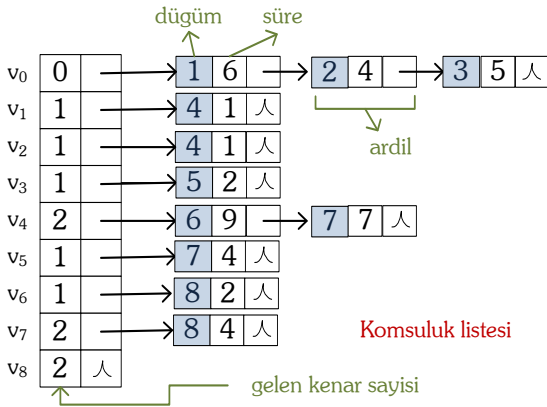
```
16:         if( EnErkenBas[k] < EnErkenBas[j] + p->süre )
17:             EnErkenBas[k] = EnErkenBas[j] + p->süre;
```

En Erken Başlama Zamanı Hesaplama Tablosu

{Başlangıç} Yığıtta, öncülü kalmamış düğümlerin numaralarını tutacağız. Topolojik sırada başta bulunan v_0 , ilk ele alınacak düğüm olduğu için, başta yığıtta bu düğümün numarası olan 0 var. TopolojikSira() yordamındaki ilk for döngüsü, bu işi yapıyor.

EnErkenBas	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Yığıt
Başlangıç	0	0	0	0	0	0	0	0	0	0
Çıktı v_0	0	6	4	5	0	0	0	0	0	3, 2, 1
v_3	0	6	4	5	0	7	0	0	0	5, 2, 1
v_5	0	6	4	5	0	7	0	11	0	2, 1
v_2	0	6	4	5	5	7	0	11	0	1
v_1	0	6	4	5	7	7	0	11	0	4
v_4	0	6	4	5	7	7	16	14	0	7, 6
v_7	0	6	4	5	7	7	16	14	18	6
v_6	0	6	4	5	7	7	16	14	18	8
v_8	0	6	4	5	7	7	16	14	18	-

EnErkenBas[] dizisini başlangıçta sıfırla doldurmalıyız. Çünkü projenin başlangıç zamanını sıfır olarak kabul ettik. Eğer 0 dışında bir sayı koysaydık, sözgelimi 10, [0] düğümünün başlama zamanı 10 kabul edilmiş olacaktı. Bu durumda diğer düğümlerin en erken başlama zamanları da bu ilk değer üzerinden hesaplanacaktı. Örneğin [1] için bulunacak değer 6 değil, $10+6=16$ olacaktı.



{ v_0 } İlk adımda v_0 'ın ardılları olan [1], [2] ve [3] düğümlerinin en erken başlama zamanları (6, 4, 5) güncellendi. Çünkü gelen değerler, mevcut değer olan 0'dan büyüktü. Bu arada bu üç düğüme **gelen kenar sayısı 1 azaltılınca 0'a düştüğü için, bu düğümler yığıta eklendi.** (Bunun neden ve nasıl olduğunu anlamadıysanız topolojik sırayı bulan yordamı anlamaya çalışmalısınız.)

{ v_3 } Sıradaki işlenecek düğüm yığıtta en başta bulunan v_3 'tür (yığıtın üstü sol taraf). v_3 'ten v_5 'e gidilebildiğinden [5] hanesine 7'yi yazdık. ([3]'te bulunan $5 + [5]$ aktivitesinin süresi 2) v_5 'in v_3 'ten başka öncülü kalmadığı için 5 numaralı düğüm de yığıta eklendi.

{ v_5 } Sıradaki düğüm olan v_5 'i ele aldığımızda tek komşusunun v_7 olduğunu görüyoruz. [0]'dan [7]'ye kadar toplam maliyet olan 11'i [7] hanesine yazdık. Fakat **bu verinin kalıcı olduğu kesin olarak söylenemez, çünkü [7]'ye gelen başka bir yol daha var. Nitekim ilerde 11 yerine 14 gelecektir.**

Peki, hangi veriler kalıcıdır? Bir düğüme gelen yolların hepsi ele alınmadıkça daha uzun bir yolla karşılaşmak muhtemeldir, bu durumda en erken başlama zamanı değişir. Öyleyse bir düğüme gelen tüm yollar ele alındıktan sonra, ancak verinin kalıcı olduğunu iddia edebiliriz. Yani, **"Ancak çıktıya yazılan düğümlerin en erken başlama zamanları kesinleşmiştir, o andan itibaren bir daha değişmez."** demiş oluyoruz. Çünkü çıktıya yazılan bir düğüm, topolojik sırada işleme sırası gelmiş bir düğümdür ki, böyle bir düğümün hiç öncülü kalmamış demektir (aksi halde çıktıya yazılmazdı). Bir düğümün hiç öncülü yoksa o düğüme giden alternatif bir yol bulmaktan da söz edilemeyeceğinden en erken başlama zamanı kesinleşmiştir.

{ v_2 } v_2 'yi aldık, komşusu olan [4]'ün sayacını 1 azalttık. **Sayaç 0'a düşmediği için [4], yığıta girmez.**

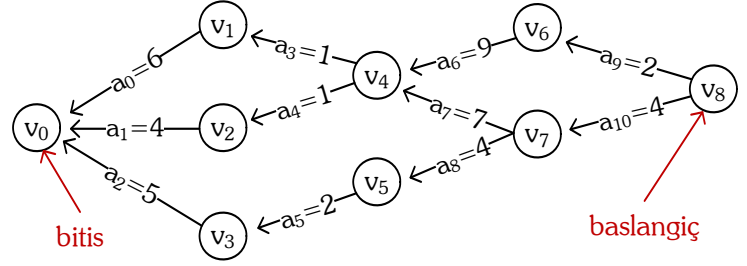
{ v_1 } Aynı minval üzere işlemlere devam ediyoruz. v_4 düğümüne v_1 üzerinden giden alternatif bir yola rastladık. **Bu yolun uzunluğu, mevcut veriden büyük olduğundan, veri güncellendi.**

{ v_4 } Öncülü kalmayan düğümlerin yığıta eklenme sırası, komşuluk listesindeki sıraya sıkı sıkıya bağlıdır. Listedeki sıra [6], [7] şeklinde olduğundan, yığıta da bu sıra ile eklendi.

...{ v_8 } Sonuçta oluşan EnErkenBas[] dizisi, her bir düğümün en erken başlama zamanını tutuyor. Projenin bitiş düğümü olan [8]'in erken başlama zamanı 18 olarak bulundu. Öyleyse **proje en erken 18 günde bitebilir.**

En Geç Başlama Zamanlarının Hesaplanması (Calculation of Latest Times)

İleri doğru (*forward*) çözümlemeyle en erken zamanları hesapladık. Şimdi de geri doğru (*backward*) çözümlemeyle en geç başlama zamanlarını bulacağız. Bunun için **tersten topolojik sırayı kullanacağız**. Dikkat! “Elimizdeki topolojik sırayı tersten kullanacağız” demiyoruz. Şunu diyoruz; **Bütün okları ters yöne çevireceğiz, bu şekliyle elde ettiğimiz çizgeden çıkan topolojik sırada çözümleme yapacağız**. Bu durumda v_8 ilk düğüm, v_0 ise son düğüm haline gelmiş olacak.



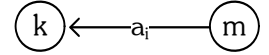
Öyleyse **çizgeyi göstermek için kullanacağımız yapı, komşuluk listesinin tersidir**. Böylece okları ters çevrilmiş yeni bir çizge elde etmiş olacağız. Devrik almanın maliyeti $O(e+n)$ 'dir.

Şu an projenin 18 günde biteceğini biliyoruz. Öyleyse v_8 olayının en geç başlama zamanı 18'dir, yoksa proje aksamış olur. Yani, **son düğüm için en erken ve en geç başlama zamanları eşittir**:

```
EnGecBas[n-1] = EnErkenBas[n-1]; // son eleman
```

v_8 'in 18. gün başlaması için ön şart olan v_7 'nin en geç 14. gün başlaması gerekir. Çünkü a_{10} aktivitesi 4 gün sürecektir. Eğer v_7 olayı 1 gün aksarsa a_{10} aktivitesi 15. gün başlar. Bu durumda v_8 düğümüne $15+4=19$. günde ancak ulaşılabilir. Bu da projenin aksadığı manasına gelir ($19>18$). Aynı şekilde v_6 olayı en geç 17. gün başlamalıdır ki, $17+1=18$. gün bitirilebilsin. Öyleyse; projenin aksamaması için **a_i aktivitesinin en geç başlama zamanı m 'nin en geç başlama zamanından a_i 'nin süresi çıkarılarak bulunabilir**:

```
EnGec(i) = EnGecBas[m] -  $a_i$  aktivitesinin süresi
```



Tıpkı en erken başlama zamanı bulurken karşılaştığımız durum gibi, burada da bir düğüme birden çok yoldan gidebilme durumuyla karşılaşabiliriz. Örneğin; En geç başlama zamanlarının v_6 için 17, v_7 için 14 olduğunu söyledik. v_6 'yı aksatmamak için v_4 , en geç $17-9=8$. gün başlamalıdır. Diğer taraftan düğümleri v_7 'yi aksatmamak için v_4 , en geç $14-7=7$. gün başlamalıdır. **İki veriden hangisini almalıyız?**

Eğer büyük olanı alırsak, “ v_4 olayı en geç 8. gün başlayabilir, 8. günü geçmedikçe proje aksamaz.” demiş oluruz. **Acaba bu doğru mudur?** v_4 olayı 8. gün başlarsa a_7 aktivitesi 7 gün süreceğinden v_7 düğümüne 15. gün ulaşılır. Hâlbuki en geç 14. gün başlaması gerektiğini söylemiştik. a_{10} aktivitesi de 4 gün süreceğinden proje $15+4=19$. güne sarkar. Demek ki, **iki veriden daha küçük olanı almalıyız**.

S(m): m düğümünün yakın ardılarının oluşturduğu küme olmak üzere $k \in S(m)$ ise;

```
EnGecBas[k] = min{ EnGecBas[m] -  $\langle m, k \rangle$  aktivitesinin süresi };
```

Öyleyse en geç başlama zamanını araştırırken, o ana kadar bulunmuş olandan daha erken bir gün bulunursa $EnGecBas[k]$ verisi güncellenmelidir. Yukarıda verdiğimiz örnekte v_4 için en geç başlama zamanı olarak 8 bilgisine daha önce rastlanmışsa $EnGecBas[4]=8$ yazılmıştır. Sonra daha erken bir günde ($7<8$) başlaması gerektiğini tespit ettiğimizde $EnGecBas[4]=7$ olarak güncellenmelidir:

```
if( EnGecBas[k] > EnGecBas[m] - p->süre )
    EnGecBas[k] = EnGecBas[m] - p->süre;
```

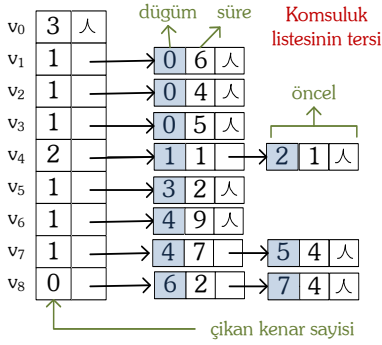
Daha önceden yazdığımız `TopolojikSira()` yordamında yapılacak küçük bir değişiklikle en geç başlama zamanını bulan yordamı da yazmış olacağız. `TopolojikSira()`'da sayac alanını azaltan 16. satırdan önceye şu kod parçasını eklememiz yeterlidir:

```
16:         if( EnGecBas[k] > EnGecBas[j] - p->süre )
17:             EnGecBas[k] = EnGecBas[j] - p->süre;
```

“Hocam nerden nereye kadar çıkacak?”

İşlediğimiz bütün konulardan gelebilir. Veri Yapıları 1'den de gelebilir. Veri yapılarını gördünüz ya artık...

En Geç Başlama Zamanı Hesaplama Tablosu



EnGecBas	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Yığıt
Başlangıç	18	18	18	18	18	18	18	18	18	8
Çıktı v ₈	18	18	18	18	18	18	16	14	18	7,6
v ₇	18	18	18	18	7	10	16	14	18	5,6
v ₅	18	18	18	8	7	10	16	14	18	3,6
v ₃	3	18	18	8	7	10	16	14	18	6
v ₆	3	18	18	8	7	10	16	14	18	4
v ₄	3	6	6	8	7	10	16	14	18	2,1
v ₂	2	6	6	8	7	10	16	14	18	1
v ₁	0	6	6	8	7	10	16	14	18	0
v ₀	0	6	6	8	7	10	16	14	18	-

{Başlangıç} Yığıta, öncülü olmayan [8] düğümünü koyarak başlıyoruz. Başlangıçta diziyi 18 ile doldurduk. Çünkü projenin tamamlanma süresi 18'dir. Geri doğru çözülemeye 18. günden başlıyoruz. Farklı bir değerle başlasaydık, sonuçta bulacağımız diğer değerler de bu değer üzerinden hesaplanacaktı. Örneğin 20 ile başlasaydık [7] için en geç başlama zamanını 18 olarak bulacaktık.

```

EnGecBas[8] = EnErkenBas[8] = 18;
{v8} EnGecBas[6] = min( EnGecBas[8]-2 ) = 16;
      EnGecBas[7] = min( EnGecBas[8]-4 ) = 14;
{v7} EnGecBas[4] = min( EnGecBas[7]-7 ) = 7;
      EnGecBas[5] = min( EnGecBas[7]-4 ) = 10;
{v5} EnGecBas[3] = min( EnGecBas[5]-2 ) = 8;
{v3} EnGecBas[0] = min( EnGecBas[3]-5 ) = 3;
{v6} EnGecBas[4] = min( EnGecBas[6]-9 ) = 7; // Değişmedi
{v4} EnGecBas[1] = min( EnGecBas[4]-1 ) = 6;
      EnGecBas[2] = min( EnGecBas[4]-1 ) = 6;
{v2} EnGecBas[0] = min( EnGecBas[2]-4 ) = 2; // Tekrar değişti
{v1} EnGecBas[0] = min( EnGecBas[1]-2 ) = 0; // Tekrar değişti

```

Bolluklar

Artık elimizde bulunan EnErkenBas ve EnGecBas dizilerini kullanarak aktivitelerin bolluklarını bulabiliriz. En geç başlama zamanından en erken başlama zamanını çıkarttığımızda o işte ne kadar bolluk olduğunu buluruz.

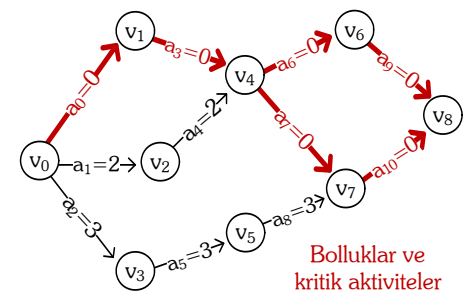
Örneğin; a₄ aktivitesinin en erken 4. gün başlayabileceğini biliyorduk, projeyi aksatmamak için en geç 6. gün başlamış olması gerektiğini de bulduk. Öyleyse a₄ aktivitesi için bolluk 2 gündür, en fazla bu kadar gecikmeye tolerans gösterilebilir.

Eğer bir aktivitenin bolluğu 0 ise, o aktivite başlayabileceği en erken vakitte başlamak zorundadır, demektir. Yoksa en geç başlama zamanı geçmiş olacaktır. Bolluğu sıfır olan bu aktivitelere **kritik aktivite** diyoruz.

Microsoft Project Manager ile proje değerlendirmenin nasıl yapıldığını görebilirsiniz. Project Manager yazılımları bolluk dağıtmayı grafiksel olarak gösterir. Kritik aktivitelerde müsamaha yok, ama diğerlerine fazladan süre verebiliriz gibi...

İki çeşit bolluk vardır. İkinci bolluk türü için yön-eylem kitaplarına bakabilirsiniz. Bizim şu an uğraştığımız bolluk türü, basit olanı (*total free*)... Eğer bu bolluğu (örneğin a₂ - a₅ - a₈ yolundaki 2 bolluğu) yol üzerindeki bir aktivite kullanırsa (a₂ veya a₅), diğerleri (a₈) kullanamaz, kritik hale gelir.

Aktivite	Kenar	En Erken	En Geç	Bolluk
a ₀	(0,1)	0	0	0: kritik
a ₁	(0,2)	0	2	2
a ₂	(0,3)	0	3	3
a ₃	(1,4)	6	6	0: kritik
a ₄	(2,4)	4	6	2
a ₅	(3,5)	5	8	3
a ₆	(4,6)	7	7	0: kritik
a ₇	(4,7)	7	7	0: kritik
a ₈	(5,7)	7	10	3
a ₉	(6,8)	16	16	0: kritik
a ₁₀	(7,8)	14	14	0: kritik



SIRALAMA (SORTING)

Bilgisayarda çoğu uygulama, elimizdeki veri yığınlarının sıralanmasını gerektirir. **Veri öğeleri (fields)** toplamından meydana gelen **tutanaklar (records)** bellek ortamı dışında tutulduklarında **kütük (file)**, bellekte tutulduklarında ise **liste** olarak isimlendirilirler.

Tutanakların '**biricik**' (**unique**) olarak tanınmasına neden olan veri öğeleri (ya da kombinasyonları) ise **anahtar (key)** bilgi olarak adlandırılır. Bir kütüğün tamamı bellek ortamına taşınarak yapılan sıralamaya **iç sıralama (internal sorting)**; disk, teyp gibi yardımcı bellek ortamları kullanılarak yapılan sıralamaya ise **dış sıralama (external sorting)** adı verilir.

Biz, iç sıralamayla ilgileneceğiz. Bu arada dış sıralama için de bir temel vermiş olacağız. Başka derslerde 10'luk bir dizi kullanarak 100'lük dosyanın nasıl sıralanacağını görüyorsunuz. Hard diske erişim süresini azaltacak parametreleri de dikkate almak gerekiyor. [[w: sorting algorithm](#)] [[v: sıralama algoritması](#)] [[sorting-algorithms.com](#)] [[carleton university](#)]

```
1: #define N 15
2: #define kucuk(A,B) (A<B)
3: #define degistir(A,B) {int t = A; A=B; B=t;}
4: #define karsilas_degis(A,B) if(kucuk(A,B)) degistir(A,B);
5: int main(int argc, char **argv) {
6:     ...
7:     ...
8:     int *a = (int*) malloc(N * sizeof(int));
9:     for (int i = 0; i < N; i++)
10:         a[i] = rand() % 50;
11:     ...
12:     selection(a, 0, N - 1); // sıralama yordamına çağrı
13:     ...
14:     free(a);
15:     ...
16:     ...
17: }
```

Seçmeli Sıralama (Selection Sort) Algoritması

```
1: void selection(int a[], int L, int R) {
2:     int i, j;
3:     for (i = L; i < R; i++) {
4:         int min = i;
5:         for (j = i + 1; j <= R; j++)
6:             if (kucuk(a[j], a[min]))
7:                 min = j;
8:         degistir(a[i], a[min]);
9:     }
10: }
```

Algoritma:

1. Listedeki en küçük değerli öğeyi bul.
2. İlk konumdaki öğeyle bulunan en küçük değerli öğenin yerini değiştir.
3. Yukarıdaki adımları listenin ilk elemanından sonrası için (2. elemandan başlayarak) yinele.

Analiz: *Selection sort*, önce listedeki en küçük elemanı seçer ve listenin başına yerleştirir. Daha sonra 2. elemandan başlayarak listedeki en küçük elemanı seçip 2. elemanla yer değiştirir. Bu şekilde her adımda, kalan alt listenin en küçük elemanını alt listenin başına yerleştirerek devam eder. En küçük elemanın seçilmesi için liste kesinlikle sona kadar taranmalıdır. Öyleyse 0 (sıfır) konumundaki eleman için listede kalan $n-1$ eleman taranır; 1 konumundaki değer için $n-2$, 2 konumundaki için $n-3$, elemana bakılmalıdır. Toplam karşılaştırma (*comparison*) sayısı:

$(n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$ olur. Bir de her adımda bulunan en küçük değerler uygun

konuma taşınacaktır. Bu da $n-1$ adet değiştirme (*exchange*) yapılacağı manasına gelir. (son elemana gelindiğinde son eleman zaten doğru konumda olacağından n değil, $n-1$) Sonuçta algoritma karmaşıklığı $O(n^2)$ olur. [\[viki\]](#)[\[wiki\]](#)

Eklemeli Sıralama (Insertion Sort) Algoritması

```
1: void insertion(int a[], int L, int R) {
2:     int i, j, v;
3:     for (i = L + 1; i <= R; i++) {
4:         // i.değer için konum ara
5:         v = a[i];
6:         // i-1 eleman sırada
7:         j = i - 1;
8:         while (j >= 0 && a[j] > v) {
9:             a[j + 1] = a[j];
10:            j--;
11:        } // while
12:        a[j + 1] = v;
13:    } // for
14: }
```

```
1: void insertion1(int a[], int L, int R) {
2:     int i;
3:     for (i = R; i > L; i--)
4:         karsilas_degis(a[i], a[i-1]);
5:     for (i = L + 2; i <= R; i++) {
6:         int v = a[i];
7:         int j = i;
8:         while (kucuk(v, a[j-1])) {
9:             a[j] = a[j - 1];
10:            j--;
11:        } // while
12:        a[j] = v;
13:    } // for
14: }
```

Soldaki *insertion* yordamının ilerlemiş hâli sağdaki *insertion1* yordamıdır.

Algoritma: Eklemeli sıralama, her döngüde girdiden **bir eleman alır** ve aldığı elemanı **sıralanmış olan listede** uygun konuma **ekler**. Girdiden alınacak eleman, her hangi bir eleman olabilir. [\[viki\]](#)[\[wiki\]](#)

Analiz:

En iyi durumda girdi sıralıdır. Bu durumda her adımda seçilen eleman, sıralı listenin yalnızca en sağındaki değer ile kıyaslanır ve daha büyük olduğu görülünce diğer değerlerle karşılaştırılmaya gerek duyulmadan listenin en sağına eklenir. Dolayısıyla en iyi durumda karmaşıklık $\Omega(n)$ 'dir.

En kötü durumda girdi tersten sıralıdır. Bu durumda her adımda seçilen eleman, listenin en solunda bulunan en küçük elemandan daha küçüktür. Seçilen eleman, sıralı listenin en soluna kadar tüm elemanlarla karşılaştırılıp en sola eklenir. Öyleyse her döngüde sıralı dizi 1 sağa kaydırılır. Dolayısıyla en kötü durumda karmaşıklık $O(n^2)$ olur.

Insertion Algoritmasının Ortalama Durum (Average Case) Analizi

En iyi ve en kötü durumu biliyoruz; ikisinin ortalamasını alınca ort. durumu analiz etmiş olur muyuz? Hayır! **Ortalama durum analizinde, örnekler uzayındaki durumların tamamını gözlememiz gerekir.**

Herhangi bir anda $[j]$ konumundaki eleman için uygun yer aradığımızı düşünelim. $[0]$ konumundan $[j-1]$ konumundaki elemana kadar bulunan j tane elemanın sıralı olduğunu algoritma garanti ediyor. $[j]$ konumundaki eleman, $[j+1]$ adet konumdan birine yerleştirilecektir.

Soldaki `insertion` yordamında $a[j] > v$ ifadesinin kaç kez doğru olacağı; sağdaki `insertion1` yordamında ise `kucuk` makrosuyla yapılan $a[i] < a[j-1]$ karşılaştırması algoritma karmaşıklığını belirleyecektir. Bunlar ise birer **rastlantı değişkenidir**; yani değeri her şey olabilir. Öyleyse algoritmayı **beklenen değer** olarak analiz edebiliriz. Bakın, bunlar istatistik ve olasılık kavramlarıdır.

C_j : $[j]$ konumundaki elemanı uygun yere taşımak için kaç karşılaştırma yapılacağını gösteren **rastgele değişken, rastlantı değişkeni (random variable)**.

$E(C_j)$: **Beklenen değer**: Yapılması tahmin edilen ortalama karşılaştırma sayısı.

Örnek çalışmamızda (dizi indisi sıfırdan başlamak üzere) $j=3$ konumundaki elemanı, 3 eleman içeren sıralı listeye ekleyeceğiz. Görüldüğü üzere olası 4 farklı durum vardır (**$j+1$ farklı duum**). Olası durumların 1'inde 1 kez, 1'inde 2 kez, 2'sinde 3 kez karşılaştırma yapılıyor. Son durumda indis taşması sebebiyle çıkıldığından dolayı bu karşılaştırmayı saymıyoruz. Böylece **son durumda da, sondan bir önceki durumla aynı sayıda karşılaştırma yapılmış olur**. Öyleyse; toplam durum sayısının $j+1$ olduğu da göz önüne alındığında tüm durumların;

$\frac{1}{j+1}$ 'inde 1 karşılaştırma, $\frac{1}{j+1}$ 'inde 2 karşılaştırma, $\frac{2}{j+1}$ 'inde 3 karşılaştırma yapılması beklenir.

Yani **tüm durumların olasılığı eşittir, yalnızca son durumun olasılığı diğerlerinin 2 katıdır**. $j=3$ için yaptığımız analizi genellersek, her hangi bir $[j]$ konumundaki değer için ortalama olarak ;

$$E(C_j) = 1 \cdot \frac{1}{j+1} + 2 \cdot \frac{1}{j+1} + 3 \cdot \frac{1}{j+1} + \dots + (j-1) \cdot \frac{1}{j+1} + j \cdot \frac{1}{j+1} + j \cdot \frac{1}{j+1} = \frac{1}{j+1} \cdot \left[2j + \sum_{k=1}^{j-1} k \right]$$

$$= \frac{1}{j+1} \cdot \left[2j + \frac{j \cdot (j-1)}{2} \right] = \frac{1}{j+1} \cdot \left[\frac{j^2 + 3j}{2} \right] = \frac{j+2}{2} - \frac{1}{j+1}$$

karşılaştırma yapılması beklenir.

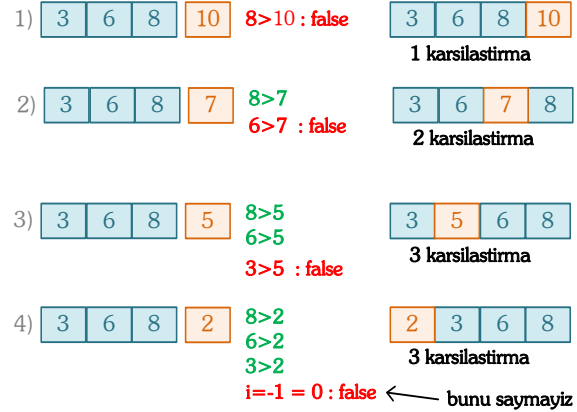
Tek bir eleman için yapılması beklenen karşılaştırma sayısını bulduk. Toplamda yapılması beklenen karşılaştırma sayısı **$Y = C_1 + C_2 + C_3 + \dots + C_{N-1}$** yazılıp hesaplanır:

$$E(Y) = E\left(\sum_{j=1}^{N-1} C_j\right) = \sum_{j=1}^{N-1} \left(\frac{j+2}{2} - \frac{1}{j+1} \right) = \left(\frac{3}{2} + \frac{4}{2} + \dots + \frac{N+1}{2} \right) - \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N} \right)$$

$$= \frac{1}{2} \left[\frac{(N+1) \cdot (N+2)}{2} - 3 \right] - \left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{N} \right) \approx \frac{N^2 + 3N}{4} - H_N$$

Bu serinin eşiti denir? Biliyor musunuz? Bu serinin alt ve üst sınırı bilinir, ama tam değeri bulunamamıştır. Alttan ve üstten log n'le sınırlıdır. Siz de hemen çözdünüz: "Hocam şudur..." ☺

$H_N = \ln n + O(\text{şekil})$ (harmonic) Sonuçta; algoritma, ortalama durumda $n^2/4$ karşılaştırma ve $n^2/4$ yarı değişim (half exchange: takas değil kaydırma) kullanır. Dolayısıyla **ortalama durumda karmaşıklık $O(n^2)$ düzeyindedir.**



Kabarcık Sıralama (*Bubble Sort*) Algoritması

```
1: void bubble(int a[], int L, int R) {
2:     int i, j;
3:     for (i = L; i < R; i++)
4:         for (j = R; j > i; j--)
5:             karsilas_degis(a[j], a[j-1]);
6: }
```

Dizi sıralıysa, hemen çıkacak şekilde daha dikkatli kodlanabilir. Örneğin; bir döngü boyunca hiç takas işlemi yapılmamışsa sonraki döngüye girmeye gerek yoktur, çünkü dizi sıralanmıştır ve artık asla takas yapılmayacaktır. [\[wiki\]](#) [\[wiki\]](#) [\[bubble sort in 28 languages\]](#) Kabarcık sıralama, en kötü durumda ve ortalama durumda $n^2/2$ karşılaştırma, $n^2/2$ değiştirme kullanır. Karmaşıklık: $O(n^2)$.

Sayarak Sıralama (*Counting Sort*) Algoritması

```
1: void counting_sort(int a[], int b[], int r, int k) {
2:     int c[K]; // K=k+1 olarak belirlenmelidir
3:     int i, j;
4:     L1: for (i = 0; i <= k; i++)
5:         c[i] = 0;
6:     L2: for (j = 0; j <= r; j++) // 0 ile k arasında değişen değerlerin
7:         c[a[j]]++;           // herbirinin sayısı nedir
8:     L3: for (i = 1; i <= k; i++) // her bir değer yerleşeceği konumunu bul
9:         c[i] += c[i - 1];
10:    L4: for (j = r; j >= 0; j--) {
11:        b[c[a[j]] - 1] = a[j]; // girdiyi yeni diziye sıralı yerleştir
12:        c[a[j]]--;           // aynı değerli başka girdi varsa bir sola yerleşecek
13:    }
14: }
```

a[] : girdi dizisi
b[] : sıralı değerler dizisi
c[] : yardımcı dizi
r : dizideki ilk r+1 adet veri sıralanacak
k : sayılar 0 ile k aralığında değişiyor

1954'te *Harold H. Seward* tarafından oluşturulan *counting sort* algoritması, **hangi aralıkta (range) olduğu bilinen bir veri grubunu sıralamada kullanılır**. Daha önce polinom çarpma probleminde anlatmıştık. İlk geçişte frekansları sayıyoruz, ikinci geçişte sıralıyoruz.

Girdilerin tamamı bir kez taranarak her değerden kaç adet bulunduğu sayılabilir. Her değerden kaç adet bulunduğu bilindikten sonra her bir değer sıralı dizide nereye konması gerektiği de bulunabilir.

Böylece değerler **birbiriyle karşılaştırılmadan** sıralanmış olur. Bu özelliği sebebiyle sayarak sıralama algoritması, *non-comparison sorts* kategorisine girer. Bu kategorideki bir diğer sıralama türü kova (*bucket*) sıralamadır. [\[wiki\]](#) [\[wiki\]](#)

Örnek; Sıralanacak veriler: 3, 0, 6, 9, 4, 3, 10, 5 olarak verilsin.

	0	1	2	3	4	5	6	7
a	3	0	6	9	4	3	10	5

İlk döngü c dizisini sıfırlar, 2. döngü her değerden kaç adet bulunduğunu sayar. Böylece c dizisi, tablodaki ilk hali alır:

	0	1	2	3	4	5	6	7	8	9	10
c	1	0	0	2	1	1	1	0	0	1	1

[3]=2: veriler içerisinde 2 tane 3 var.

[7]=0: veriler arasında hiç 7 sayısı yok.

3. döngü, her değerin yerleşmeye başlayacağı konumları bulur:

	0	1	2	3	4	5	6	7	8	9	10
c	1	1	1	3	4	5	6	6	6	7	8

[9]=7: 9'dan küçük ya da 9'a eşit 7 eleman vardır. 9 gelirse 7-1=6. konuma taşı.

[2]=1: 2'den küçük ya da 2'ye eşit 1 eleman vardır. 2 gelirse 2-1=1. konuma taşı. (2 yok)

[0]=1: 0'dan küçük ya da 0 eşit 1 eleman vardır. 0 gelirse 1-1=0. konuma taşı.

Son döngüde, veriler küçükten büyüğe sıralı olarak b dizisine yazılır:

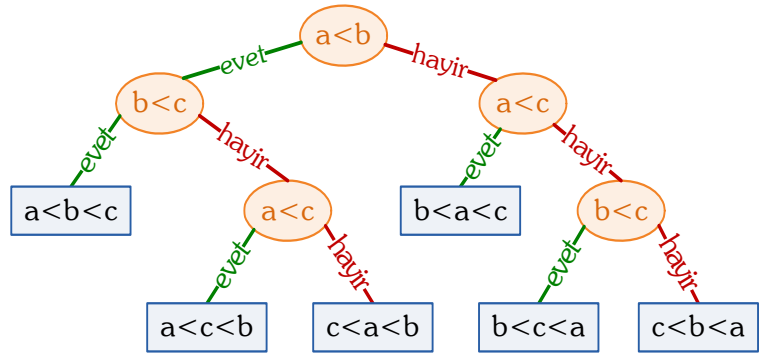
	0	1	2	3	4	5	6	7
b	0	3	3	4	5	6	9	10

Analiz: L1: O(k) L2: O(n) L3: O(k) L4: O(n)

Karmaşıklık O(n+k) olur. Eğer k, n'li bir fonksiyonsa o zaman O(n) diyebiliriz. Doğrusal zamanda sıralama yapmak için, veri aralığını bilmeliyiz. 10 değer sıralanacaksa (n=10); eğer aralık 0 ile 1000 arasında değişiyorsa n³'lük algoritma olur. Aralık 1-100 ise n²'lik, 1-50 ise n²/2'lik, 1-25 ise n²/4'lük fonksiyon olur. Dolayısıyla **aralığın (range) çok küçük olması lazımdır. Bir diğeri de paket sıralama (bir tane bağlı liste). Hızlı devrik alma tek bellekle olur demiştik ya, işte bu o...**

Optimal Sıralama Zamanı (Optimal Sorting Time)

Her hangi bir veri türünden oluşan n elemanlı bir listeyi en çok ne kadar hızlı sıralayabiliriz? Örnek olarak şekildedeki **karar ağacı (decision tree)** {a,b,c} değerlerini sıralamamıza yardımcı olacaktır. Sınama ifadeleri elipslerle, karar ifadeleri ise dikdörtgenlerle temsil edilmiştir.



Bir karar ağacında uç yaprakların sayısı n! olur. n=3 için çizilen ağaçta uç yaprak sayısı 3! = 6 olmuştur.

Teorem: n tane farklı elemanı sıralayan her hangi bir karar ağacının derinliği en az $\log_2(n!) + 1$ olarak verilir.

Derinlikle ilgileniyoruz, çünkü kaç karşılaştırma yapılacağı derinlikle ilgilidir. Genel bir sıralama algoritmasının en kötü durumda en az $\lceil \log(n!) \rceil$ adet karşılaştırma yapma gereksinimi vardır. n!'in belli bir eşiti yok, [Stirling katsayısı](#) ile alt ve üst sınır verilebiliyor:

$$\lceil \log(n!) \rceil = n \cdot \lg n - n \cdot \lg e + \Theta(\lg n) \approx n \cdot \lg n - 1,4427 \cdot n$$

Bundan sonra yazacağımız algoritmaları karşılaştırmak için bir **mihenk taşı** verdik.

Hızlı (Çabuk) Sıralama (Quick Sort) Algoritması

1962'de **Sir Charles Antony Richard Hoare** tarafından geliştirilmiştir. (Tony Hoare olarak da tanınır, 1934 doğumlu, hala yaşıyor, hala üretiyor). [viki](#)[wiki](#)

Çabuk sıralama algoritması yalnızca küçük bir yardımcı yığıt (*auxiliary stack*) kullanır. n adet veriyi sıralamada ortalama olarak $n \cdot \lg n$ ile orantılı bir zaman ister. Quicksort, kararlı (*stable*) olmamakla birlikte, en kötü durumda (*in worst case*) n² işlem gerektirse de bilinen en hızlı algoritmalarından biridir.

Performans analizleri üzerinde fazlaca çalışma yapılmıştır. Girdinin dağılımına bağlı olarak performansını artırmak için düzenlenmiş uyarlamaları (*versions*) da vardır. Standart C kütüphanesinde **qsort** hazır fonksiyonu bulunur.

Quick sort algoritması, **parçala ve yönet (divide and conquer)** yaklaşımını kullanır. Bu yaklaşımda problem, doğrudan çözülebilecek kadar küçülene dek özyineli olarak iki ya da daha fazla alt probleme bölünür.

```

1: void quicksort1(int a[], int L, int R) {
2:     int bol(int*, int, int);
3:     int i;
4:     if (R <= L)
5:         return;
6:     i = bol(a, L, R);
7:     quicksort1(a, L, i - 1);
8:     quicksort1(a, i + 1, R);
9: }
10:
11: int bol(int a[], int L, int R) {
12:     int i = L - 1, j = R, v = a[R]; // pivot eleman a[R] olarak seçildi
13:     for (;;) {
14:         while (kucuk(a[++i], v))
15:             ;
16:         while (kucuk(v, a[--j]))
17:             ;
18:         if (j == L)
19:             break;
20:         if (i >= j)
21:             break;
22:         degistir(a[i], a[j]);
23:     }
24:     degistir(a[i], a[R]);
25:     return i;
}

```

Not: Horowitz'in kitabındaki kod yanlış olabilir. Bu kod, Sedgewick'in kitabından alınmıştır. Oldukça güzel yazılmış bir kod. ([Sedgewick](#), [Knuth](#)'un doktora öğrencisidir; *quick sort* üzerinde fazlaca çalışmıştır.)

[6:] *Quick sort* algoritmasında genel strateji, sıralanacak diziyi bölme mantığına dayalıdır. O yüzden öncelikle dizinin hangi noktadan bölüneceği tespit edilir.

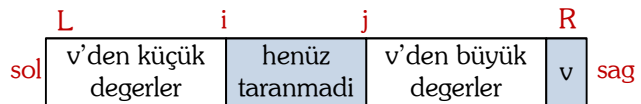
[12:] Bölme elemanı (*partitionary element*) olarak önce $a[r]$ seçilir ve değeri v değişkenine konur.

[14:] Daha sonra soldan (L) dizinin sonuna doğru bölme elemanından daha büyük ya da eşit bir eleman bulana dek ilerleriz. $kucuk(a[++i], v)$ ifadesinde ilk yapılacak işlem i 'nin artırılması olduğundan, bu tarama tam olarak L konumundan başlar ve pivot elemandan daha büyük ya da pivot elemana eşit ilk değerle karşılaşınca sonlanır.

Başlangıçta bölme elemanı olarak seçtiğimiz v değerinin dizinin en sonunda bulunduğunu hatırlayalım. Bu yüzden, bu döngüde **indis taşmasını kontrol etmemiz gerekmez**. Çünkü en kötü ihtimalle $i=R$ olduğunda $a[i]$ değeri v 'ye kesinlikle eşit olacaktır ve döngüden çıkılacaktır. Yani döngüdeki kontrol ifadesi indis taşması olmayacağını dolaylı olarak garanti etmektedir.

[16:] Sağdan (R) da dizi başına doğru, bölme elemanından daha küçük ya da eşit bir eleman buluncaya kadar ilerleriz. [17:] Önceki döngüden farklı olarak burada **indis taşması durumunu kontrol etmek durumundayız**. Çünkü döngüdeki kontrol ifadesi [16:] indis taşması olmayacağını garanti edemez.

[21:] Taramayı sağdan ve soldan sonlandıran bu iki elemanın konumları, bölümlenmiş diziyeye göre açıkçası yanlıştır. Bu nedenle yerleri değiştirilir (*exchange*).



Böylece sol göstergenin (i) solundaki hiçbir değer bölme elemanından daha büyük değildir. Sağ göstergenin (j) sağındaki hiçbir değer de bölme elemanından daha küçük değildir. [19:] Diziyi sağ ve sol göstergeler ortada birleşene kadar [13:] `for` içindeki bu bölümleme süreci devam eder.

[23:] `for` döngüsünden çıkıldığında ilk başta seçtiğimiz pivot eleman için doğru konumu bulmuş oluruz; artık pivot elemanı buraya yerleştirebiliriz. **Bu, öyle bir konumdur ki;** solundaki değerler kesinlikle pivot elemandan daha küçük ya da eşit, sağdaki değerler de kesinlikle pivot elemandan daha büyük ya da eşittir. Öyleyse **pivot eleman doğru konuma taşınmıştır**. Yani veriler sıralansaydı, pivot eleman tam olarak bu konuma yerleşecekti.

Değerlendirme: Algoritma “Veriler sıralansaydı $a[R]$ elemanı burada olurdu.”yu garantiledi. Diyeceksiniz ki, kabarcık ya da seçmeli sıralamada da her adımda bir eleman için doğru konum bulunuyordu. **Evet! Onlar da buluyor, ama problemi küçülterek sonraki adıma katkı yapamıyorlar.** Seçmeli sıralamada sonraki adımda alt problem $n-1$ olarak kalıyor. Ama hızlı sıralama, hem bir elemanı doğru konuma yerleştiriyor, hem de problemi küçülterek sonraki adıma katkı yapıyor. İşte 1960’larda keşfedilen algoritma bu...

Quick sortla ilgili üç-dört algoritma yazacağım. O yüzden bunu anlamanız önemli. Şimdi, kitaptakinden farklı bir analiz yapacağım. Neden en hızlı algoritmalarından biri olarak biliniyor? O, şimdi anlatacağım mantıkta gizli:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
37	15	24	33	49	11	51	77	72	48
37	15	24	33	11	49	51	77	72	48
37	15	24	33	11	48	51	77	72	49
11	15	24	33	37	48	49	77	72	51
11	15	24	33	37	48	49	51	72	77
11	15	24	33	37	48	49	51	72	77
11	15	24	33	37	48	49	51	72	77
11	15	24	33	37	48	49	51	72	77

OKUMA PARÇASI: Algoritma ve Bilgisayar

— “Hocam! Adam bu quick sort’u 1960’larda nerde denemiş, nasıl test etmiş?”

Algoritma kelimesi nereden geliyor? 8. yüzyılda el-Harezmi’nin 4GB’lık laptopu mu vardı? ☺

O zamanlar henüz b^2-4ac keşfedilmemiş. Harezmi’nin 2. dereceden bir bilinmeyenli denklemin köklerinden bir tanesini bulan algoritması var. **Algoritma demek; hiç bilmeyen adam onu alacak, adım adım işletecek ve doğru sonucu bulacak, demek.**

Kaldı ki, 1960’larda bilgisayar zaten vardı. Fakat **bizim dersimiz, bilgisayardan bağımsızdır.** Görüyorsunuz ya; ben kodu tahtada yazıyorum, bilgisayarda işletmiyorum. Zaten bir algoritmanın makalesinin çıkması yeterlidir.

El-Harezmi (780-850)

Ebu Abdullah Muhammed bin Musa el-Harezmi; matematik, gökbilim, astroloji, coğrafya ve haritacılık alanlarında çalışmış ünlü bir Fars bilginidir. 780 yılında Harzem bölgesinin Hive şehrinde (Özbekistan) dünyaya gelmiştir. 850 yılında Bağdat’ta vefat etmiştir. Hayatının çoğunu Bağdat’taki Beytu’l-Hikmet’te bilim adamı olarak çalışarak geçirmiştir.

Yerkürenin **bir derecelik meridyen yayı uzunluğunu ölçmek** için Sincar Ovası’na giden bilim heyetinde bulunmuştur. Bugünkü bilgisayar bilimi ve dijital elektroniğin temeli olan **2’lik (binary) sayı sistemi** ve **0 (sıfır)** buldu. 12. yüzyılda **Arap rakamlarını açıklayan Arithmetic** adlı eseriyle **onluk konumsal sayı sistemi**ni ve **sıfırı Batı dünyasına tanıttı.**

820’li yıllarda yazdığı *El’Kitab’ül-Muhtasar fi Hesab-il Cebir ve’l-Mukabele* (Lâtinçe: *Algebra*, Cebir ve Denklem Hesabı Üzerine Özet Kitap) adlı kitabı, matematik tarihinde, birinci ve ikinci dereceden denklemlerin sistematik çözümlerinin yer aldığı ilk eserdir. Bu eser aynı zamanda doğu ve batının ilk müstakil cebir kitabı olma özelliğini taşımaktadır. Bu nedenle Harezmi (*Diophantus* ile birlikte) **"cebirin babası"** olarak bilinir.

İngilizce’deki *"algebra"* ve bunun Türkçe’deki karşılığı olan **"cebir"** sözcüğü, Harezmi’nin kitabındaki ikinci dereceden denklemleri çözmeye yöntemlerinden biri olan *"el-cebr"*den gelmektedir. **Algoritma** (*algorithm*) sözcüğü de Harezmi’nin Latince karşılığı olan *Algoritmi*’den türemiştir ve yine İspanyolca’daki basamak anlamına gelen *guarismo* kelimesi Harezmi’den gelmiştir. [\[viki\]](#) [\[viki\]](#)

Worst case: Eğer dizi sıralı ise *quick sort* yordamı kendi kendini n kez çağıracaktır. Her fonksiyon çağırısında pivot eleman, bulunduğu konumda kalır ve sonraki çağrılar dizi boyutu bir eksiltilecek yapılır.

$$n + (n-1) + \dots + 2 + 1 = \frac{n \cdot (n+1)}{2} = O(n^2) \text{ Şimdi "Bunun neresi hızlı?" diye düşüneceksiniz.}$$

Best case: Algoritma, en iyi durumda (*best case*) her adımda diziyi tam olarak ortasından böler. Sadece ilk seferde değil, her seferde... Her adımda i , $n/2$ kez artar; j , $n/2$ kez azalır; **toplamları n olur. Sonraki adıma ise $n/2$ büyüklüğünde alt problemler kalır.** Bu durumda algoritma karmaşıklığını ifade edecek özyineli ilişki $C_n = 2C_{n/2} + n$ olarak verilebilir. Bu denklemin çözümü için farklı yöntemler uygulanabilir (*recursion*, *iteration*, *substitution*, tümevarım) Biz, en kolay olan *iteration* yöntemiyle çözeceğiz:

$$C_n = 2C_{n/2} + n = 2 \cdot \left(2 \cdot C_{\frac{n}{2^2}} + \frac{n}{2} \right) + n = 2^2 \cdot C_{\frac{n}{2^2}} + n + n = 2^k \cdot C_{\frac{n}{2^k}} + k \cdot n = 1 \cdot n \cdot \lg n$$

Karşılaştırma sayısı bunun altına düşmez, $n \cdot \lg n$ alt sınırdır. Daha önceden buduğumuz optimal sıralama ile karşılaştırdığımızda bunun da katsayısı 1, ama $1.44n$ fazlalık var. Yanında eğer $-1.44n$ bulsaydık, en hızlı olacaktı. Bu algoritma optimal sıralamadan $1.44n$ daha fazla karşılaştırma yapar.

Quick Sort Algoritmasının Ortalama Durum (Average Case) Analizi

Şimdi en kötü ve en iyi durum analizlerini yaparak alt ve üst sınırları belirledik. $n \cdot \lg n$ 'den az olmaz, n^2 'den de çok olamaz. Bu ikisi arasında bir yerlerde olabilir. **Peki, nerede yoğunlaşır?**

Yazı tura attın, kaç kez yazı geleceğini araştırıyorsun. 10 kez, 100 kez, 1000 kez, hatta 1 milyon kez attın. Biri dedi ki; $\frac{1}{2}$ 'ye yakınsar! Vay be! İşte, şimdi onu göstereceğiz. Nasıl göstereceğiz? Hadi göster, at bakalım! $\frac{1}{2}$ geliyor mu? İşte, **nerede yoğunlaşacağını ortalama durum analizi yaparak saptarız.**

Bizim dersimizde olasılık çok önemlidir. Yurt dışına giden doktora öğrencileri "Ya hocam, biz istatistik bilmiyormuşuz." diyorlar. Rastgele sayılar üretecek, nasıl üretecek? Windows'ta *uniform* üretmiyoruz, bu hususta UNIX daha sağlam.

Gelelim analyze: Algoritma, diziyi tam ortadan bölmeyebilir. Olası bölme noktaları $1 \leq k \leq n$ arasında değişebilir. Eğer bölme, k . konumda meydana gelirse, bu durumda $(k-1)$ ve $(n-k)$ eleman içeren iki alt problemin, iki alt dizinin sıralanması gerekecektir.

"Bana göre, k 'nın hepsinde olma ihtimali eşittir." **Bana göresi, sana göresi yok bu işin! Matematiği var bu işin.** Olası $n!$ adet farklı diziliş vardır, hepsi için $\text{böl}()$ yordamını ayrı ayrı çağırınca frekanslar eşit oluyorsa, " k 'nın hepsinde olma ihtimali eşittir." varsayımı doğrudur. Öyleyse [insertion sort algoritmasının ortalama durum analizi](#)nde yaptığımız gibi ortalama karşılaştırma sayısını bulmak için rastlantı değişkeni C_n 'i hesaplayacağız.

$C_1 = C_0 = 0$ ve $n \geq 2$ olmak üzere karşılaştırma sayıları dikkate alınarak şu özyineli ilişki verilebilir:

$$C_n = (n+1) + \frac{1}{n} \cdot \sum_{1 \leq k < n} (C_{k-1} + C_{n-k})$$

$n+1$, $\text{böl}()$ yordamındaki bölme elemanını karşılaştırma maliyetidir. if karşılaştırmalarını saymanın pek anlamı yok, ama onu da 1 kabul ettik. Bölme, herhangi bir konumda bulunabilir. O yüzden, alt problemler toplamını n 'e bölerek ortalamasını aldık.

$$\text{Sigma (toplam) sembolünü açtığımızda; } \sum_{1 \leq k < n} (C_{k-1} + C_{n-k}) = \sum_{k=1}^n C_{k-1} + \sum_{k=1}^n C_{n-k} = 2 \cdot \sum_{k=0}^{n-1} C_k \text{ olur.}$$

Yani $C_0 + C_1 + \dots + C_{n-1}$ toplamı iki kez görüleceğinden $C_n = (n+1) + \frac{2}{n} \cdot \sum_{k=1}^n C_{k-1}$ yazılır. Burdan;

$$n \cdot C_n = n \cdot (n+1) + 2 \sum_{k=1}^n C_{k-1} \dots\dots\dots (1) \text{ eşitliği elde edilir.}$$

Kitap, bunu yerine koymayla çözüyor. Biz daha akıllıca bir taktikle çözeceğiz. n yerine n-1 koyalım;

$$(n-1) \cdot C_{n-1} = n \cdot (n-1) + 2 \sum_{k=1}^{n-1} C_{k-1} \dots\dots\dots (2) \text{ Fibonacci üzerinde uğraşmış olmalısınız: } f_n = f_{n-1} + f_{n-2}$$

Şimdi (1)'den (2)'yi çıkarıyoruz.

$$\begin{aligned} n \cdot C_n - (n-1) \cdot C_{n-1} &= n \cdot (n+1) + 2 \sum_{k=1}^n C_{k-1} - \left(n \cdot (n-1) + 2 \sum_{k=1}^{n-1} C_{k-1} \right) \\ &= n \cdot (n+1) - n \cdot (n-1) + 2 \cdot \left(\sum_{k=1}^n C_{k-1} - \sum_{k=1}^{n-1} C_{k-1} \right) \\ &= 2n + 2 \cdot C_{n-1} \end{aligned}$$

$$n \cdot C_n = 2n + (n+1) \cdot C_{n-1}$$

$$C_n = \frac{n+1}{n} \cdot C_{n-1} + 2 \text{ Böylece denklemi iterasyon yöntemiyle çözülecek hale getirdik.}$$

$$\begin{aligned} C_n &= \frac{n+1}{n} \cdot \left(\frac{n}{n-1} \cdot C_{n-2} + 2 \right) + 2 = \frac{n+1}{n-1} \cdot C_{n-2} + \frac{2 \cdot (n+1)}{n} + 2 \\ &= \frac{n+1}{n-1} \cdot \left(\frac{n-1}{n-2} \cdot C_{n-3} + 2 \right) + \frac{2 \cdot (n+1)}{n} + 2 \\ &= \frac{n+1}{n-2} \cdot C_{n-3} + \frac{2 \cdot (n+1)}{n-1} + \frac{2 \cdot (n+1)}{n} + 2 = \frac{n+1}{n-2} \cdot \left(\frac{n-2}{n-3} \cdot C_{n-4} + 2 \right) + \frac{2 \cdot (n+1)}{n-1} + \frac{2 \cdot (n+1)}{n} + 2 \\ &= \frac{n+1}{n-3} \cdot C_{n-4} + \frac{2 \cdot (n+1)}{n-2} + \frac{2 \cdot (n+1)}{n-1} + \frac{2 \cdot (n+1)}{n} + 2 \\ &= \dots \end{aligned}$$

$$= \frac{n+1}{3} \cdot C_2 + \frac{2 \cdot (n+1)}{4} + \frac{2 \cdot (n+1)}{5} + \frac{2 \cdot (n+1)}{6} + \dots + \frac{2 \cdot (n+1)}{n-1} + \frac{2 \cdot (n+1)}{n} + 2$$

$$= 0 + 2 \cdot (n+1) \cdot \left[\frac{1}{4} + \frac{1}{5} + \dots + \frac{1}{n} \right] + 2. \text{ Bu denklemden, } \sum_{k=1}^n \frac{1}{k} = \ln n + O(1) \text{ olduğundan;}$$

$$C_n \leq 2 \cdot (n+1) \cdot \ln n \approx 2 \cdot n \cdot \ln n = 2 \cdot n \cdot \ln n \cdot \frac{\ln 2}{\ln 2} = 2 \cdot \ln 2 \cdot n \cdot \frac{\ln n}{\ln 2}$$

$$\approx 1,39 \cdot n \cdot \lg n = O(n \cdot \lg n) \text{ bulunur.}$$

Şu an algoritmanın her yerine değilse de, çoğuna hâkimiz. **Direnç noktalarını tespit ettik.** Ama bitmedi! Dedik ki; “Özyineli çağrılar sebebiyle az da olsa bir yığıt kullanıyor.” Sizin kullandığınız standart C fonksiyonu olan `qsort()` daha hızlı çözüyor. Ne yapıyor acaba?

Özyinelemesiz Quick Sort Algoritması

Hızlı sıralama algoritmasını özyinelemesiz (*non-recursive*) yazarak, sistem yığıtı kullanımını azaltmayı amaçlıyoruz.

```
1: #define yigita_at(A,B) { yigit_ekle(B); yigit_ekle(A); }
2:
3: void quicksort2(int a[], int L, int R) {
4:     void yigit_baslat(void);
5:     int yigit_bosmu(void);
6:     void yigit_ekle(int);
7:     int yigit_al(void);
8:     int bol(int*, int, int);
9:
10:    int i;
11:    yigit_baslat();
12:    yigit_at(L, R);
13:
14:    while (!yigit_bosmu()) {
15:        L = yigit_al();
16:        R = yigit_al();
17:        if (R <= L) continue;
18:
19:        i = bol(a, L, R);
20:        if (i - L > R - i) {
21:            yigita_at(L, i - 1);
22:            yigita_at(i + 1, R);
23:        } else {
24:            yigita_at(i + 1, R);
25:            yigita_at(L, i - 1);
26:        }
27:    }
28: }
```

[20,23:] if-else sayesinde sol problem ($i-L$) ve sağ problemden ($R-i$) hangisi daha büyükse, yığta önce o konur, sonra da küçük parça konur. Bir sonraki adımda yığta son eklenen parça geri alınıp işleneceği için önce küçük parça alınmış olur. Böylece yığtta n tane küçük parçanın birikmesi engellenir. Artık yığt dolmayacaktır, **yığta 1 eleman girip çıkacaktır.**

[15:] İki alt listeden küçük olan önce sıralanırsa, n elemanlı bir diziyi sıralamak için yığt büyüklüğü $\lg n$ değerini hiçbir zaman aşamaz. Yığta ilişkin özyineli ilişki (*space complexity*):

$$T_n = T_{\frac{n}{2}} + 2 \text{ (binary search algoritmasının karmaşıklığıyla aynıdır), } T_1 = T_0 = 0.$$

$$= \left(T_{\frac{n}{2^2}} + 2 \right) + 2$$

$$= T_{\frac{n}{2^2}} + 2 \cdot 2$$

$$= T_{\frac{n}{2^k}} + 2 \cdot k$$

$$n = 2^k \text{ olduğundan } k = \lg n \text{ olur. Öyleyse } T_n = T_1 + \lg n = \lg n$$

Eğer if bloğu koyarak yığt kullanımını optimize etmeseydik, ancak her seferinde ortadan bölerse bellek karmaşıklığı bu olurdu.

Küçük kodları defalarca farklı şekillerde yazmalısınız.

Geliştirilmiş Quick Sort Algoritması

```
1: void quicksort3(int a[], int L, int R) {
2:     int bol(int*, int, int);
3:     int i;
4:     if (R - L <= M)
5:         return;
6:     degistir(a[(L+R)/2], a[R-1]);
7:     karsilas_degis(a[L], a[R-1]);
8:     karsilas_degis(a[L], a[R]);
9:     karsilas_degis(a[R-1], a[R]);
10:    i = bol(a, L + 1, R - 1);
11:    quicksort3(a, L, i - 1);
12:    quicksort3(a, i + 1, R);
13: }
14:
15: void sirala(int a[], int L, int R) {
16:     quicksort3(a, L, R);
17:     insertion(a, L, R);
18: }
```

C'deki standart kütüphane fonksiyonu olan `qsort()` muhtemelen budur.

Quicksort algoritması en kötü performansını, **insertion** algoritması ise en iyi performansını sıralı veri grubunda gösteriyordu.

Hızlı sıralamanın bu versiyonu, **3 medyanlı bölümlendirme** (*median-of-three partitionary*) kullanır. 7, 8 ve 9. satırlarda bulunan 3 adet aktarma komutu, en kötü durumu (sıralı olma durumunu) bozar. Böylece $n \lg n$ 'den biraz daha fazla karşılaştırma yaparak problemi çözer. Seni asla n^2 ile karşı karşıya getirmez. Analizi biraz ağır...

`sirala()` yordamında bulunan `quicksort3()` çağrısından döndüğünde M büyüklüğünde kendi içinde sırasız parçalar kalabilir. Kalan bu parçalar da eklemeli sıralama ile sıralanır. Eğer M değeri, $5 \leq M \leq 25$ arasında ise bu yaklaşım, işletim süresini çok fazla değiştirmeksizin **%10 iyileştirme** sağlamaktadır.

`sirala()` yordamını kullanmadan bu yaklaşımı uygulamak istersek, 4 ve 5. satırları şu şekilde değiştirmemiz yeterli olacaktır:

```
4:     if (R - L <= M) { insertion(a, L, R);
5:         return; }
```

OKUMA PARÇASI: Zorunlu Stajlar

Stajlar, öyle zannediyorum ki, imkânsızlıktan dolayı konmuş olmalı. Eskiden programı yazıyordunuz, getirip laboratuarda kartlara deldiriyordunuz. Bir kişiye günde bir, bilemedin iki kez ancak derleme sırası geliyordu. Sonra da programın çıktısını alıp hatayı bulmaya çalışıyordunuz. Yüksek lisans yapanlar, ancak gece 12'den sonra çalışabiliyordu, çünkü gündüzleri kalabalık oluyordu. Şimdi sizin günde kaç kez derleme imkânınız var? 50 mi, 100 mü? Kaç? Hemen herkesin laptopu var, değil mi? İşte o yüzden, "Öğrenciler bari gitsinler de farklı şirketlerde, başka bilgisayarlarda çalışıp kendilerini geliştirme imkânı bulsunlar." diye staj zorunlu hale getirilmiş olmalı.

Şimdi ise durum farklı... Bizim öğrenciyi şunu yap, bunu yap diye ayak işlerinde kullanıyorlar. Klavyeyle toprak kazdırmak gibi... Öğrenci staja gidiyor, ama kendini geliştiremiyor ki! İşin alaveresini dala-veresini öğreniyor, üçkâğıdını öğrenip geliyor. İş ortamını öğreniyor, sonra 3. sınıfta *part-time* çalışmaya başlıyor. "Ya piyasa dolarsa!" diye korkuyor. Böylelerinin 2 yıllık önlisans diplomasını bir an önce alıp, hemen piyasaya atılmasında fayda var. ☺

Bizim piyasayla işimiz yok ki! Bizim için önemli olan, bilim adamı yetiştirmek... *Mustafa Ege*

Birleştirmeli Sıralama (Merge Sort) Algoritması

1945 yılında *John von Neumann* [w][v] tarafından bulunan *merge sort* algoritması, *divide and conquer* yaklaşımının kullanıldığı diğer bir sıralama algoritmasıdır.

Algoritma: [wiki][wiki]

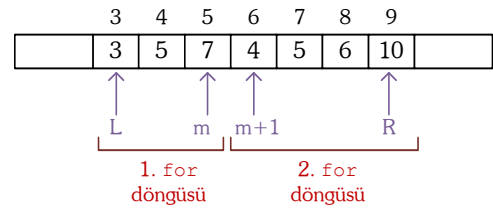
1. Liste 0 ya da 1 elemanlı ise zaten sıralıdır.
2. Değilse, listeyi olabildiğince birbirine yakın boyutta iki parçaya ayır.
3. Her alt listeyi merge sort'u özyineli olarak uygulamak suretiyle sırala.
4. Kendi içinde sıralı olan iki alt listeyi tekrar tek bir sıralı listede birleştir.

Birleştirmeli sıralama, çalışma zamanını iyileştirmek için **şu iki fikir üzerine kuruludur:**

1. Küçük bir listeyi sıralamak, büyük bir listeyi sıralamaktan daha az adımda yapılabilir.
2. Sıralı iki alt listeden bir sıralı liste elde etmek, sırasız iki alt listeden bir sıralı liste elde etmekten daha az adımda yapılabilir.

```
1: void mergesort(int a[], int L, int R) {
2:     void merge(int*, int, int, int);
3:     int m = (R + L) / 2;
4:     if (R <= L) return;
5:     mergesort(a, L, m);
6:     mergesort(a, m + 1, R);
7:     merge(a, L, m, R);
8: }
9:
```

```
10: void merge(int a[], int L, int m, int R) {
11:     // Bu yordam a[L] <= a[L+1] <= ... <= a[m] ve a[m+1] <= a[m+2] <= ... a[R]
12:     // olduğunu kabul ederek yardımcı b dizisi aracılığıyla
13:     // a[L] <= ... <= a[R] olacak biçimde yerleştirir
14:     int i, j, k; // L <= m <= R
15:     for (i = m+1; i > L; i--) b[i - 1] = a[i - 1];
16:     for (j = m; j < R; j++) b[R+m-j] = a[j + 1];
17:     for (k = L; k <= R; k++)
18:         if (kucuk(b[j], b[i]))
19:             a[k] = b[j--];
20:         else
21:             a[k] = b[i++];
22: }
```



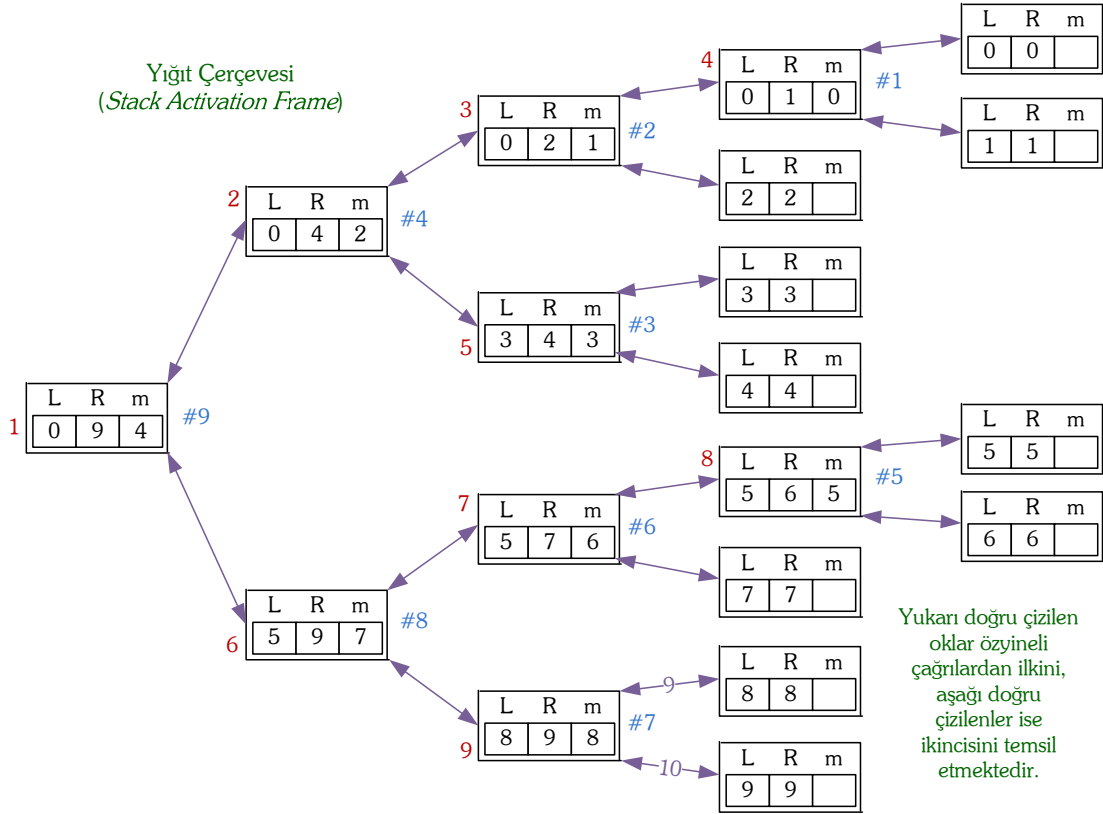
[1:] mergesort() yordamı, basitçe, algoritmada ifade edilen adımları uygular. [6:] Sol ve [7:] sağ alt listeleri kendi içlerinde sıralayacak olan özyineli çağrılarını yapar. [8:] Ardından kendi içlerinde sıralı olan iki alt listeyi tek bir sıralı listede birleştirecek olan merge() çağrısını yapar.

[11:] Verilerin sıralı hâle gelmesinde esas faktör merge() işlevidir. merge() işlevi, sıralı iki alt listeden sıralı tek bir liste elde eder. a dizisini sıralı hâle getirirken yardımcı bir b[] dizisi kullanılır. Yukarıdaki kodun derlenebilmesi için a[] ile aynı boyutta bir b[] dizisinin ya genel, ya yerel olarak tanımlanması; yahut da yordama parametre olarak geçirilmesi gerekir.

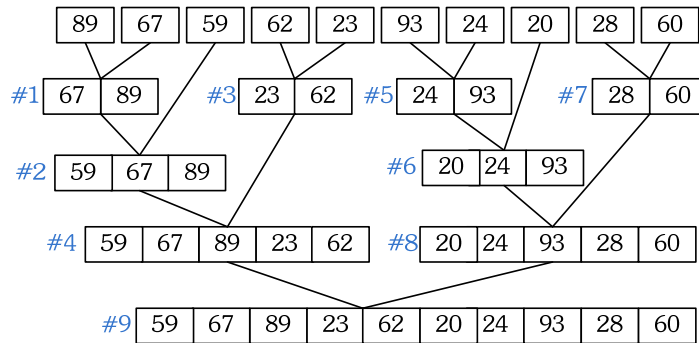
[18:] Öncelikle a[] dizisinin sol yarısı, b[] dizisinin sol yarısına birebir kopyalanır. Bu işlemdeki döngü değişkeni i'nin değeri; i=L olduğunda döngüden çıktığına dikkat ediniz. [20:] Daha sonra a[] dizisinin sağ yarısı, b[] dizisinin sağ yarısına bu kez ters sırada kopyalanır. Bu işlemdeki döngü değişkeni j'nin değeri; j=R olduğunda döngüden çıktığına dikkat ediniz. Sonuçta b[] dizisinin sol başından ortasına doğru ilerlendiğinde de, sağ başından ortasına doğru gidildiğinde de değerlerin küçükten büyüğe sıralı olduğu görülür.

[22:] i ve j'nin 3. döngüde kesinlikle indis taşmasına uğramaz. Çünkü son döngüye gelindiğinde i=L ve j=R değerlerini içermektedir; her döngüde yalnızca bir öge işleneceğine göre döngüden çıktığında i ve j değerlerinin birbirine eşit olacağını söyleyebiliriz. Dolayısıyla i ve j için sınır kontrolleri olmadan yazılmış bu kod, doğru çalışır. Bu son döngüde yapılan işlem, sıralı iki alt diziyi iki baştan tara-
yarak 'merge' etmektir.

Mademki girdiyi ortadan ikiye bölmek marifet, işte bu da ortadan bölüyor. Ama öncekilerde pivot eleman bir adımda doğru konumuna yerleşiyordu; bunda ise bir adımda doğru konuma yerleşmiyor. Sıralı iki diziden 'sıralı olma' avantajını kullanarak yeni bir sıralı dizi elde ediyor. Veri Yapıları 1 dersinde 'Bu mantığa merge diyoruz.' diye hep söylemişimdir.



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]		
89	67	59	62	23	93	24	20	28	60	anahtar	
67	89	59	62	23	93	24	20	28	60	#1	(10)
59	67	89	62	23	93	24	20	28	60	#2	(7)
59	67	89	23	62	93	24	20	28	60	#3	(7)
23	59	62	67	89	93	24	20	28	60	#4	(10)
23	59	62	67	89	24	93	20	28	60	#5	(4)
23	59	62	67	89	20	24	93	28	60	#6	(1)
23	59	62	67	89	20	24	93	28	60	#7	(1)
23	59	62	67	89	20	24	28	60	93	#8	(4)
20	23	24	28	59	60	62	67	89	93	#9	(10)



Soru: Yordamdaki iki mergesort satırının yerleri değiştirilirse nasıl bir tablo oluşur, gözlemleyiniz.

Merge Sort Algoritmasının Analizi

Zaman Karmaşıklığı

a_n : n elemanlı dizi için **toplam karşılaştırma sayısı** olarak tanımlansın. Dizi, $\lfloor n/2 \rfloor$ ve $\lceil n/2 \rceil$ boyutlarında iki parçaya ayrılır. Örneğin 16 veri varsa 8+8 şeklinde ayrılırken 17 veri varsa 8+9 şeklinde ayrılacaktır. `merge()` ise n kez çağrılacağından a_n için şu eşitlik yazılabilir:

$$a_n = a_{\lfloor \frac{n}{2} \rfloor} + a_{\lceil \frac{n}{2} \rceil} + n$$

$a_1 = 0$ 'dır. $n = 2^k$ kabul edilirse $a_n = a_{2^k} = a_{2^{k-1}} + a_{2^{k-1}} + 2^k = 2a_{2^{k-1}} + 2^k$ Sonuçta;

$$a_n = \Theta(n \cdot \lg n) \text{ bulunur.}$$

Bellek Karmaşıklığı

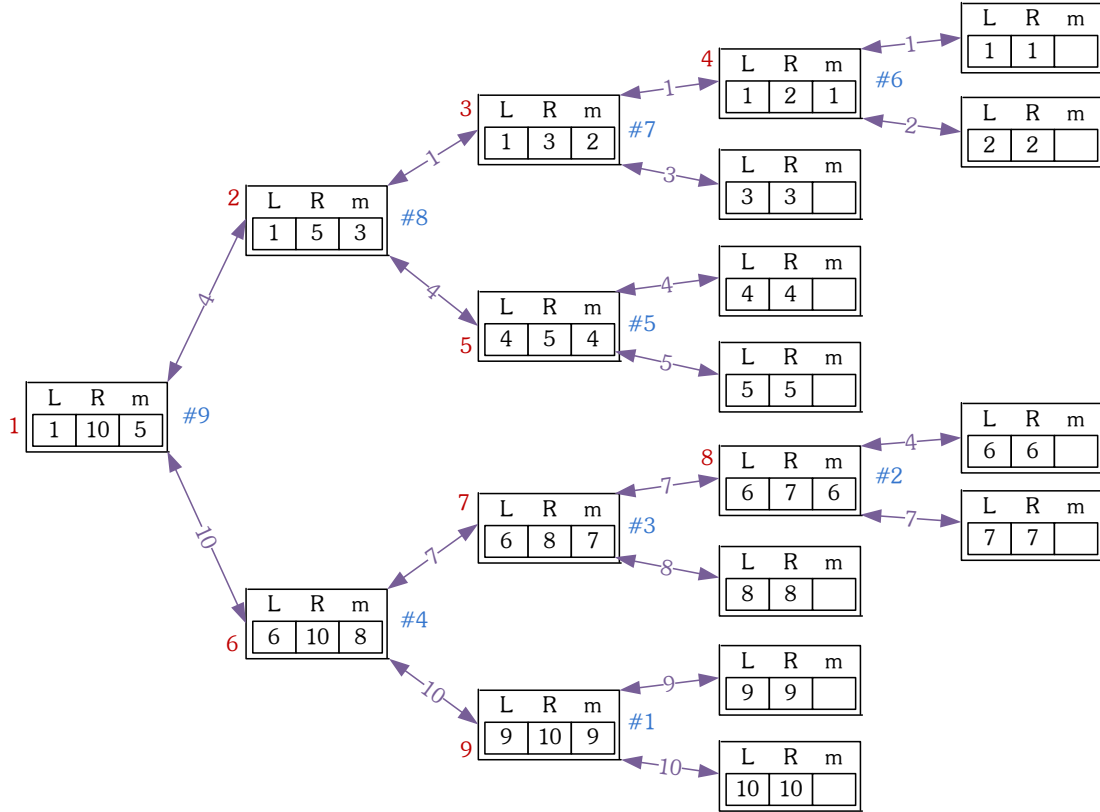
Dizi durumuna bağlı olmaksızın n elemanlı bir diziyi sıralama zamanı $n \cdot \lg n$ ile orantılıdır. *Mergesort* algoritmasının dezavantajı ise sıralamada $O(n)$ 'lik fazladan bellek istemesidir. Fazladan bellek istemesi sorunu aşılabılır; fakat bunu yapmak, işlem maliyetini artırır. Bu durumda alternatif sıralama algoritması olarak (yine $O(n \cdot \lg n)$ maliyetli olan) *heapsort* algoritması ön plana çıkar. *Mergesort* algoritması, *quicksort* ve *heapsort* algoritmalarına göre daha kararlı bir algoritmadır.

$$\begin{aligned} T_n &= 2T_{\frac{n}{2}} + 1 = 2 \left(2T_{\frac{n}{2^2}} + 1 \right) + 1 = 4 \cdot T_{\frac{n}{2^2}} + 2 + 1 = 4 \cdot \left(2T_{\frac{n}{2^3}} + 1 \right) + 2 + 1 = 8 \cdot T_{\frac{n}{2^3}} + 4 + 2 + 1 \\ &= 2^k \cdot T_{\frac{n}{2^k}} + \sum_{i=0}^{k-1} 2^i = n \cdot T_0 + \frac{n \cdot (n+1)}{2} \end{aligned}$$

Alfabetik Verileri Sıralamak

```
1: typedef struct {
2:     int anahtar;
3:     // diğer alanlar
4:     int bag;
5: } eleman;
6: void mergesort(int a[], int L, int R) {
7:     void merge(int*, int, int);
8:     int m = (R + L) / 2;
9:     if (R <= L) return;
10:    return merge(a, mergesort(a, L, m), mergesort(a, m + 1, R));
11: }
12: void merge(int a[], int birinci, int ikinci) {
13:     int sonuc = 0;
14:     for (int i1 = birinci, i2 = ikinci; i1 && i2;) {
15:         if (a[i1].anahtar <= a[i2].anahtar) {
16:             a[sonuc].bag = i1;
17:             sonuc = i1;
18:             i1 = a[i1].bag;
19:         } else {
20:             a[sonuc].bag = i2;
21:             sonuc = i2;
22:             i2 = a[i2].bag;
23:         }
24:     }
25:     if (i1 == 0)
26:         a[sonuc].bag = i2;
27:     else
28:         a[sonuc].bag = i1;
29:     return a[sonuc].bag;
30: }
```

Eğer mergesort algoritmasıyla sayıları değil de alfabetik bilgiler içeren verileri sıralamak istersek, zaman alıcı bir kopyalama işlemiyle karşı karşıya kalırız. Bu sorunun üstesinden gelebilmek için bağlaçlı bir yapı kullanabiliriz. Başlangıçta bütün bağ alanlarında sıfır değeri bulunacaktır. Sıralanması istenen veriler dizinin ilk indisi dışındaki verilerdir. $a[0]$ elemanı ise geçici bir takas alanı olarak kullanılacaktır.



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]		
71	77	122	38	119	41	22	115	43	13	anahtar	
0	0	0	0	0	0	0	0	0	9	#1	(10)
0	0	0	0	0	0	6	0	0	9	#2	(7)
0	0	0	0	0	8	6	0	0	9	#3	(7)
0	0	0	0	0	9	6	0	8	7	#4	(10)
0	0	0	5	0	9	6	0	8	7	#5	(4)
2	0	0	5	0	9	6	0	8	7	#6	(1)
2	3	0	5	0	9	6	0	8	7	#7	(1)
2	5	0	1	3	9	6	0	8	7	#8	(4)
2	8	0	0	3	9	4	5	1	7	#9	(10)

Şimdi merge algoritmasının #4 ve #8 ile işaret edilen alt listeleri $\{a[1...5], a[6...10]$ sıralı iken} nasıl birleştirdiğini görelim. (#9 işleminin nasıl gerçekleştirildiğini veriyoruz):

[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
71	77	122	38	119	41	22	115	43	13
2	5	0	1	3	9	6	0	8	7
2	8	0	0	3	9	4	5	1	7

38	71	77	119	122	13	22	41	43	115
----	----	----	-----	-----	----	----	----	----	-----

Yığın Sıralama (Heap Sort) Algoritması

Yığın sıralamanın algoritma karmaşıklığı da mergesort gibi $O(n \cdot \lg n)$ 'dir. Fakat fazladan bellek kullanmaması yığın sıralamanın bir artısıdır. [\[wiki\]](#)[\[wiki\]](#)

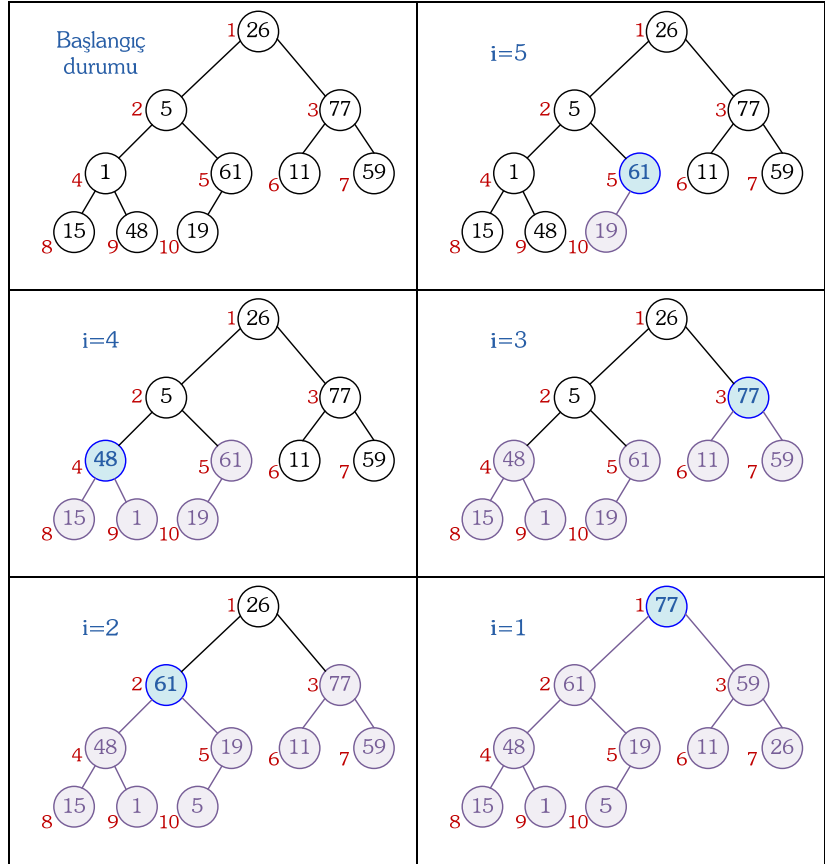
Elimizde n elemanlı; ilk elemanına 1 indisiyle, son elemanına n indisiyle eriştiğimiz bir dizi olsun. (0 nolu indisi kullanmıyoruz) Söz konusu dizi, 'tam ikili ağaç' gibi düşünülerek, yığın özelliği taşıyan bir ağaç biçimine dönüştürülebilir.

```
1: void heapify(int a[], int kok, int n) {
2:     int t;
3:     t = a[kok];
4:     for (int j = 2*kok ; j <= n; j *= 2) {
5:         if (j < n) // sonraki satırdaki j+1 için indis taşması kontrolü
6:             if (a[j] < a[j + 1]) // büyük çocuğun j'de olması sağlandı
7:                 j++;
8:         if (t >= a[j]) // t, arada bir yere girebilir
9:             break;
10:        a[j / 2] = a[j]; // atasıyla yer değiştir (ataya kaydır)
11:    }
12:    a[j / 2] = t;
13: }
14: void heapsort(int a[], int n) {
15:     void heapify(int*, int, int);
16:     // dizinin yığın haline getirilmesi
17:     for (int i = n / 2; i >= 1; i--)
18:         heapify(a, i, n);
19:     // (buraya yazılacak kod parçası ilerleyen sayfalarda verilecek)
20:     .....
21: }
```

[1:] heapify() yordamı, kok ile işaret edilen düğümü kök kabul eden ağacı maksimum öncelikli yığın haline getirir. kok'un sağ ve sol alt ağaçlarının zaten maksimum öncelikli yığın yapısına uyduğu kabul edilir. Yani heapify() yordamı, yığın yapısına uymayan tek değer olan kok'teki değer için ağaçta uygun konum arar; yoksa tüm ağacı yığın haline getirmeyi sağlar. Sonuçta, kok ile işaret edilen alt ağaç maksimum öncelikli yığın yapısına uygun hale gelmiş olur.

[19:] Ağacın tamamını yığın yapısına çevirmek için, heapify() yordamı sondan başa doğru tüm düğümler için sırayla işletilmelidir. Böylece alt ağaçlar, uçlardan köke doğru parça parça yığın yapısına uygun hale gelecektir. En nihayet, tüm ağaç bir yığın olacaktır.

Fakat heapify() yordamını uç düğümler için işletmeye gerek yoktur. Çünkü uç düğümler zaten tek bir düğüm olduğu için, alt ağaçları bulunmadığı için, yığın yapısına uymamaları gibi bir durum söz konusu olamaz. Öyleyse en az bir alt ağacı bulunan düğümlerden işletmeye başlamalıyız ki, bunlar da $\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor - 1$, $\lfloor n/2 \rfloor - 2$, ..., 3, 2 ve 1. düğümlerdir.



Kısacası; tamamen rastgele verilerle oluşturulmuş bir dizi üzerinde, `heapify()` yordamını üstte bahsedilen tüm düğümler için ($\lfloor n/2 \rfloor$, $\lfloor n/2 \rfloor - 1$, $\lfloor n/2 \rfloor - 2$, ..., 3, 2 ve 1) sondan başa doğru sırayla işletip 1 indisli yere ulaştığımızda maxheap elde etmiş oluruz.

Zaman Karmaşıklığı

`heapsort()` yordamının karmaşıklığını bulmak için öncelikle `heapify()` yordamının karmaşıklığını bilmeliyiz.

Kaba analiz: `heapify()` yordamında 12. satırdaki atama ifadesi en çok $\lg n$ kez işletilebilir ($\text{kok}=1$ ile çağrıldığı durumda). `heapsort()` yordamında 18. satırdaki döngünün karmaşıklığı ise n ile orantılıdır. Öyleyse maxheap yapma maliyeti $n \cdot \lg n$ olur. Fakat bu, kaba bir analizdir. Çünkü her çağrının $\lg n$ olduğunu farz etmiş oluyoruz. Gerçekte yalnızca 1 ile çağrılırsa $\lg n$ olur. Diğer durumlarda bundan az olur. Daha detaylı bir analiz yaptığımızda hem `heapsort()` yordamının karmaşıklığını, hem de Kruskal'ın gerçekleştiriminde minheap kurmanın maliyetini bulmuş olacağız.

İnce analiz: `heapify()` yordamının karmaşıklığı, kok düğümünün bulunduğu düzeye bağlıdır. 'Tam ikili ağaç'ta derinlik (düzey sayısı), $\lg n$ idi, buna (düzey sayısına) k diyelim. Ayrıca ikili ağaçta i . düzeyde en fazla 2^{i-1} tane düğüm bulunabiliyordu. Öyleyse `heapify()` yordamındaki döngüde yapılabilecek en fazla kaydırma sayısı şu şekildedir:

$$\sum_{i=1}^k 2^{i-1} (k-i) \quad \rightarrow \text{Bu ifadeyi düzenleyelim;}$$

$$\begin{aligned} \sum_{i=1}^k 2^{i-1} (k-i) &= \sum_{i=1}^k 2^{i-1} \cdot k - \sum_{i=1}^k 2^{i-1} \cdot i = k \cdot \sum_{i=1}^k 2^{i-1} - \sum_{i=1}^k 2^{i-1} \cdot i = k \cdot \sum_{i=1}^k 2^{i-1} - \sum_{i=1}^k 2^{i-1} \cdot i \\ &= k \cdot [2^0 + 2^1 + 2^2 + \dots + 2^{k-1}] - [2^0 \cdot 1 + 2^1 \cdot 2 + 2^2 \cdot 3 + \dots + 2^{k-1} \cdot k] \end{aligned}$$

Elde ettiğimiz sonucun k ortak çarpanı bulunan sol tarafı bir geometrik seridir, onu biliyoruz:

$$S(y) = 1 + y + y^2 + \dots + y^k = \frac{y^{k+1} - 1}{y - 1}$$

Fakat sağ tarafı bilmiyoruz, o halde analizini nasıl yapacağız? $S(y)$ 'nin türevini alalım:

$$\frac{d(S(y))}{d(y)} = 1 + 2y + 3y^2 + \dots + k \cdot y^{k-1} = \frac{(k+1) \cdot y^k \cdot (y-1) - 1 \cdot (y^{k+1} - 1)}{(y-1)^2} \quad \rightarrow \text{Şimdi } y=2 \text{ koyalım:}$$

$$\begin{aligned} 1 + 2 \cdot 2 + 3 \cdot 2^2 + \dots + k \cdot 2^{k-1} &= \frac{(k+1) \cdot 2^k \cdot (2-1) - 1 \cdot (2^{k+1} - 1)}{(2-1)^2} \\ &= (k+1) \cdot 2^k - (2^{k+1} - 1) = k \cdot 2^k + 2^k - 2^{k+1} + 1 \\ &= k \cdot 2^k + 2^k - 2 \cdot 2^k + 1 = k \cdot 2^k - 2^k + 1 \end{aligned}$$

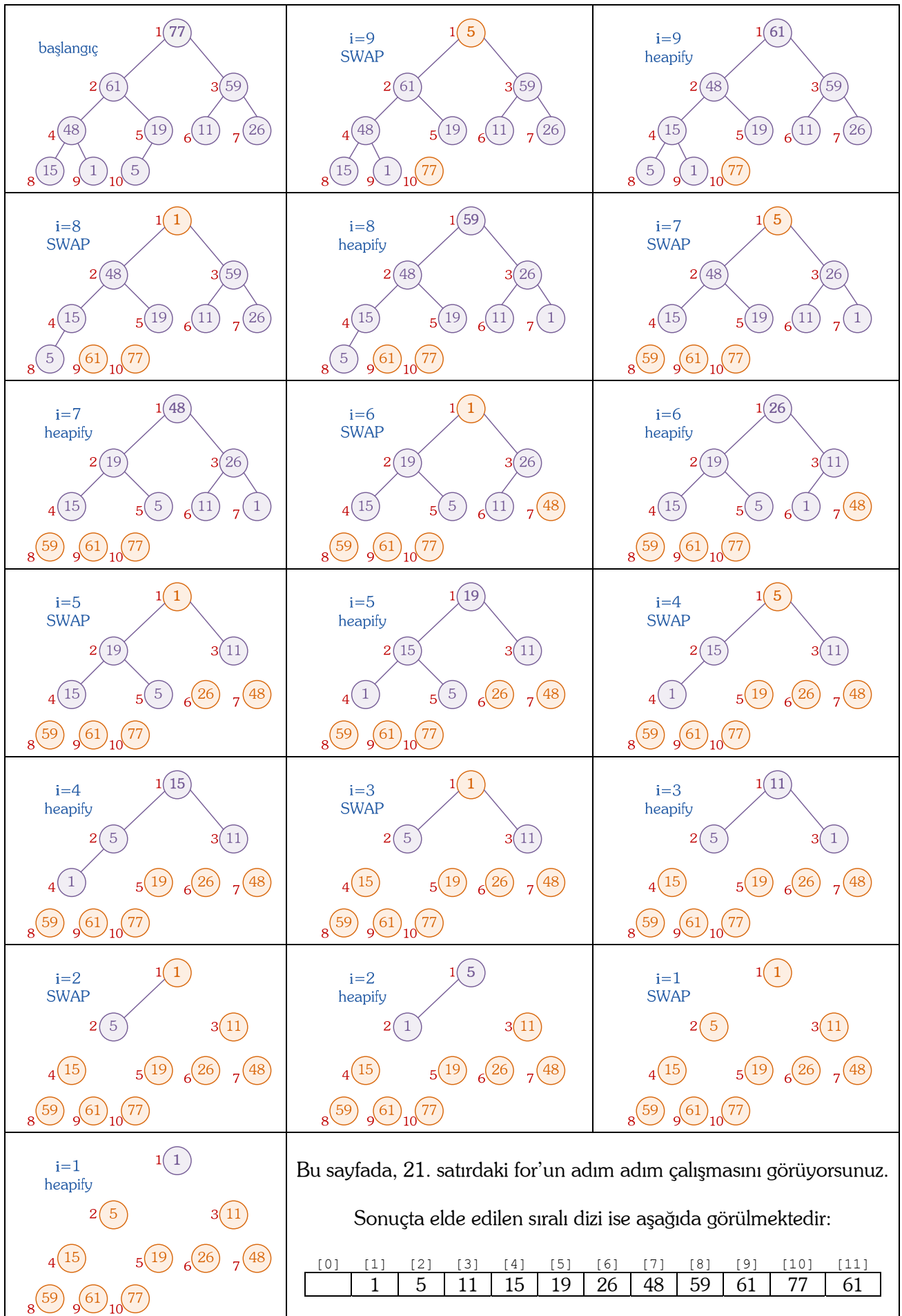
Elde ettiğimiz eşitlikleri, baştaki (`heapify` yordamındaki for için yazılan) ifadede yerine yazalım:

$$\sum_{i=1}^k 2^{i-1} (k-i) = k \cdot \frac{2^k - 1}{2 - 1} - (k \cdot 2^k - 2^k + 1) = k \cdot 2^k - k - k \cdot 2^k + 2^k - 1$$

$$= 2^k - k - 1 \quad \rightarrow k = \lg(n+1) \text{ diyebiliriz.}$$

$$= 2^{\lg(n+1)} - \lg(n+1) + 1 \quad \rightarrow \text{Bundan daha büyük bir şey yazacaksak, } 2n \text{ yazabiliriz.}$$

$$\leq 2n = o(n) \quad \rightarrow \text{Hiçbir zaman } 2n \text{'e varmaz, küçük oh notasyonu.}$$



Kruskal'ı gerçekleştirirken, elemanları tek tek minheap'e ekleyerek yığın oluşturmştunuz. O şekilde gerçekten heap oluşur, doğrudur. Ama elemanların bulunduğu diziyi, onun **yarısı kadar** bir maliyetle minheap haline getirdik.

Tamam hocam! Onu anladık da, bu dizi ne zaman sıralanacak, onu anlamadık. Evet, şimdi yukarıdaki kodda boş bıraktığımız yere aşağıdaki kod parçasını ekleyerek sıralama işlemini gerçekleştirmiş olacağız.

```
19: // dizinin sıralanması (heapsort'ta boş bırakılan yere yazılacak)
20: int p;
21: for (i = n - 1; i >= 1; i--) {
22:     SWAP(a[1], a[i + 1], p); // ağaçtaki en büyük değeri sona taşı
23:     heapify(a, 1, i);        // bozulan yığın yapısını düzelt
24: }
```

Algoritma: Kalan ağaçtaki en büyüğün kökte olduğuna emin ol, sonra ağacın sonuna taşı.

`heapsort()`: $2n \lg n + 2n = O(n \lg n)$???

İleri Heapsort Algoritmaları:

50.000'den daha az sayıda veri için **heapsort** makul bir sıralama algoritmasıdır. Karmaşıklığı $2n \cdot \lg n$ 'dir.

$n \geq 50000$ içinse bunun bir ilerisi quicksort'tan da daha hızlı olan **bottom-up heapsort**'tur. İki karşılaştırmayı teke düşürüyor, ortalama durumda algoritma karmaşıklığı $1.25n \cdot \lg n$ oluyor.

Bundan da hızlısı ise iki Çinli'nin bulduğu **optimal heapsort** algoritmasıdır. Karmaşıklığı $1 \cdot n \cdot \lg n$ 'dir. Katsayıyı 1'e düşürüyor, teorik olarak en hızlı, ama içinde ekstra bazı işlemler bulunduğuundan pratikte ancak 1.600.000 elemandan sonra performansını göstermeye başlıyor.

Counting sort örneğinde olduğu gibi, sıralanacak verilere ait bazı özellikleri biliyorsak, ona göre bir yaklaşım seçebiliriz. Bu bağlamda **shell sort ve package sort'un birlikte kullanımları** var. **Siz temel birkaç tanesini bilmelisiniz.**

Basamağa Göre Sıralama (Radix Sort) Algoritması

Buraya kadar anlatılan sıralama algoritmalarında sadece bir anahtara göre sıralama söz konusu edildi. Fakat **birden çok anahtara göre sıralamak** isteyebileğimiz veriler de olabilir. **Örneğin** ad ve soyadlardan oluşan bir kişi listesini öncelikle soyadına göre, soyadı aynı olanları ise kendi içlerinde ada göre alfabetik olarak sıralamak isteyebiliriz.

Anahtarlar (*keys*), $K^0, K^1, K^2, \dots, K^{r-1}$ şeklinde gösterilirse, K^0 **en anlamlı anahtar** (*most significant key*), K^{r-1} ise **en az anlamlı anahtar** (*least significant key*) olarak isimlendirilir. Örneğimizde 2 adet anahtar alan vardır; 'soyadı' alanı en anlamlı, 'ad' alanı ise en az anlamlı anahtardır.

Basamak sıralama algoritması, üstte anlatılan yaklaşımı **tamsayıları sıralamak için** kullanır. Öncelikle **en az anlamlı basamağa** (*LSD: least significant digit*) göre tüm verileri sıralar. Daha sonra sırayla tüm anahtarlara göre sıralama işlemini yapar; en sonunda **en anlamlı basamağa** (*MSD: most significant digit*) göre sıralama yapar. Bu noktada veriler, **LSD Radixsort** algoritmasına göre **sıralanmış olur**.

Basamak sıralama, sayıları birbiriyle karşılaştırmadan sıraladığı için **karşılaştırmaz** (*non-comparison*) **sıralama algoritmaları** sınıfına girer. [\[wiki\]](#)[\[wiki\]](#)

```

1: #define MAX_DIGIT 3
2: #define RADIX_SIZE 10
3: typedef struct oge *oge_gost;
4: typedef struct oge {
5:     int deger[MAX_DIGIT]; // 179 değeri {1, 7, 9} ile gösterilir
6:     oge_gost bag;
7: };
8:
9: oge_gost radixsort(oge_gost p) {
10:     oge_gost on[RADIX_SIZE], arka[RADIX_SIZE];
11:     int i, j, basamak;
12:
13:     for (i = MAX_DIGIT - 1; i >= 0; i--) {
14:         for (j = 0; j < RADIX_SIZE; j++)
15:             on[j] = arka[j] = NULL;
16:         while (p) {
17:             basamak = p->deger[i];
18:             if (!on[basamak])
19:                 on[basamak] = p;
20:             else
21:                 arka[basamak]->bag = p;
22:             arka[basamak] = p;
23:             p = p->bag;
24:         }
25:         p = NULL;
26:         for (j = RADIX_SIZE - 1; j >= 0; j--) {
27:             if (on[j]) {
28:                 arka[j]->bag = p;
29:                 p = on[j];
30:             }
31:         }
32:     }
33:     return p;
34: }

```

K^0 : yüzler basamağı (MSD)

K^1 : onlar basamağı

K^2 : birler basamağı (LSD)

Bu yordam, pozitif tamsayılardan en fazla 3 basamaklı olanları ([0,999]) sıralayabilir. Sayının her basamağını ayrı bir anahtar olarak değerlendirebilmek için, **her basamağı bir dizide ayrı ayrı hanelerde tutuyoruz**. Örneğin 179 sayısını $deger = \{1, 7, 9\}$ şeklinde tutuyoruz.

Karmaşıklık MAX_DIGIT ve $RADIX_SIZE$ değerlerine bağlıdır:

$$O(MAX_DIGIT \cdot (RADIX_SIZE + n + RADIX_SIZE)) = O(MAX_DIGIT \cdot (RADIX_SIZE + n))$$

Basamak sıralamanın sıralayacağı veriler (a) pozitif olmalı, (b) tamsayı olmalı ve (c) aralığı da küçük olmalıdır. Sadece tamsayıları sıralayabilmesi bir **dezavantajdır**. Kayan noktalı sayı türleri için uyarlanabilir, ama bu uyarlamayı yapana kadar diğer algoritmalar sıralamış olur. Eğer aralık büyük olursa sıralama yine gerçekleşir, ama **algoritma karmaşıklığı hızla artar**. Aşağıda verilecek örnekte karmaşıklık $O(3 \cdot (10 + n))$ olur, **lineer gibi gözüküyor**, ama *counting sort* gibi özel bir durumdur.

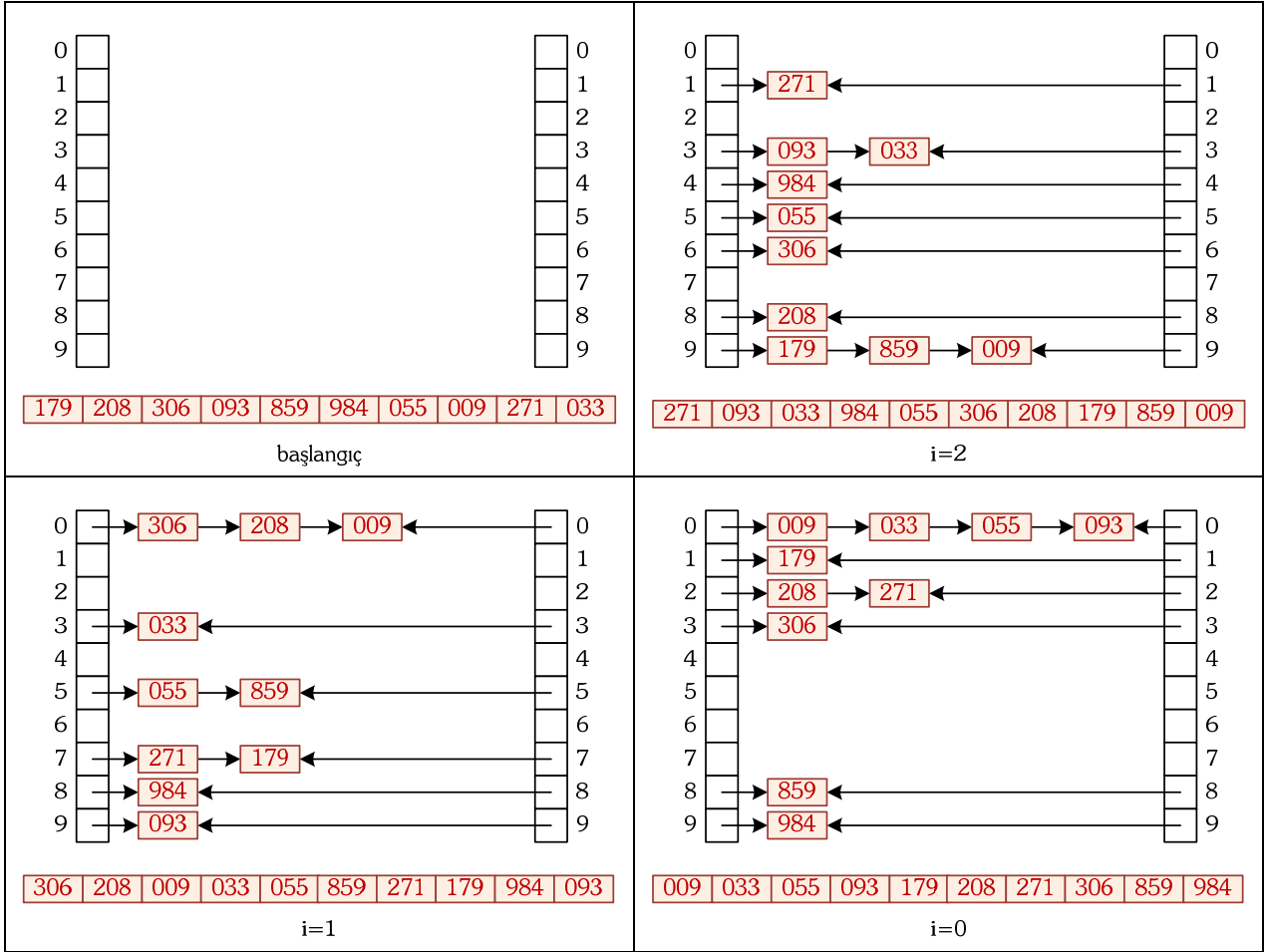
Örnek: Sıralanacak değerler sırasıyla 179, 208, 306, 93, 859, 984, 55, 9, 271, 33 olarak verilsin. Başlangıçta elimizdeki bağlı liste şu şekildedir ve her adımda oluşan yeni liste gösterilmiştir:

Başlangıç: $p \rightarrow 179 \rightarrow 208 \rightarrow 306 \rightarrow 093 \rightarrow 859 \rightarrow 984 \rightarrow 055 \rightarrow 009 \rightarrow 271 \rightarrow 033$

$i=2$: $p \rightarrow 271 \rightarrow 093 \rightarrow 033 \rightarrow 984 \rightarrow 055 \rightarrow 306 \rightarrow 208 \rightarrow 179 \rightarrow 859 \rightarrow 009$

$i=1$: $p \rightarrow 306 \rightarrow 208 \rightarrow 009 \rightarrow 033 \rightarrow 055 \rightarrow 859 \rightarrow 271 \rightarrow 179 \rightarrow 984 \rightarrow 093$

$i=0$: $p \rightarrow 009 \rightarrow 033 \rightarrow 055 \rightarrow 093 \rightarrow 179 \rightarrow 208 \rightarrow 271 \rightarrow 306 \rightarrow 859 \rightarrow 984$



UTANSIN!

Tohum saç, bitmezse toprak utansın!
Hedefe varmayan mızrak utansın!

Hey gidi küheylan, koşmana bak sen!
Çatlarsan, doğuran kısrak utansın!

...

Ustada kalırsa bu öksüz yapı,
Onu sürdürmeyen çirak utansın!

...

N.F.K.

SINAVLARDA ÇIKMIŞ SORULAR

2009 Quiz – 1. Şube

```
1: typedef enum boolean { FALSE, TRUE };
2: typedef struct oge *oge_gost;
3: typedef struct oge {
4:     boolean isaret;
5:     union {
6:         char veri;
7:         oge_gost L_bag;
8:     }
9:     int sayac; // sayaç alanı eklendi
10:    oge_gost bag;
11: };
```

Yukarıdaki struct tanımı verildiğinde; geliştirilmiş listede işaret alanı TRUE olan her öğenin sayaç alanına ilgili öğenin altında yer alan öğelerin sayısını bulup yerleştiren ve ana yordama da toplam öğe sayısını döndüren yordamı özyineli olarak yazınız.

Öğrenci Çözümü:

```
1: int oge_say( oge_gost L ){
2:     int miktar = 0; // boş liste 0 öge içerir
3:     for(;L; L = L->bag ){
4:         if( L->isaret ){ // alt liste var mı
5:             // sayaca alt listede bulunan toplam eleman sayısını yaz
6:             L->sayac = oge_say( L->L_bag );
7:             // alt listenin elemanları bu listenin de elemanıdır
8:             miktar += L->sayac;
9:         }
10:        miktar++; // şu an gezilen elemanı da say
11:    }
12:    return miktar;
13: }
```

2009 Quiz – 2. Şube

Bellekte bulunan geliştirilmiş listeyi doğrusal formda yazdıran özyineli yordamı yazınız.

Öğrenci Çözümü:

2009 Ara Sınav 1

Değerlerin tekrarlı olmadığı bir ikili ağaçta (*binary tree*) x değerini içeren ögenin ağırlığını bulmak istiyoruz. Ağırlık, “uç düğümlere kadar olan yolların en uzununu” olarak tanımlanır. Örneğin verilen ağaçta 2’nin ağırlığı 4, 17’nin ağırlığı 1, ... dir. Eğer ağaçta x değerini içeren bir öge yoksa ya da $L=NULL$ ise ağırlık sıfır olarak dönmelidir.

- Çözüm önerinizi anlaşılır bir biçimde açıklayınız.
- Daha sonra `int AğırlıkBul(agac_gost L, int x)` yordamını kodlayınız.

Bu soruda tasarım gücünüz test edilmektedir. Çözümünüzde global değişken kullanmayın. Eğer kullanacaksanız `yığıt_ekle()` ve `yığıt_al()` yordamlarının yerel (*local*) prototiplerinin verilmesi yeterlidir. Eğer başka yordamlar kullanacaksanız hem yerel (*local*) prototiplerinin hem de yordamların ayrıntılı olarak verilmesi gerekmektedir.

Kodlarınızın yapısal, okunabilir ve istenilen özelliklere sahip olmasına özen gösteriniz.

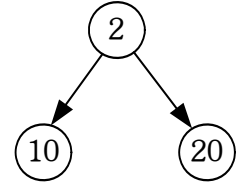
Not: Çözümünüz işletilmeden anlaşılamıyor ise tasarımınız hatalıdır.

Öğrenci Çözümü:

```
1: int AğırlıkBul( agac_gost L, int x ){
2:     // agacta_ara:L ağacında x değerini içeren öğeye gösterge döndürür
3:     // x değerini içeren öge bulunamazsa ya da L NULL'sa NULL döndürür
4:     agac_gost agacta_ara( agac_gost L, int x );
5:
6:     // derinlik: L ağacının derinliğini döndürür
7:     int derinlik( agac_gost L );
8:
9:     // bulunursa x'in ağırlığını, bulunamazsa sıfır döndür
10:    return derinlik( agacta_ara( L, x ) );
11: }
12:
13: agac_gost agacta_ara( agac_gost L, int x ){
14:     if(!L) return NULL;
15:     if( L->veri == x ) return L;
16:
17:     agac_gost T = agacta_ara( L->solcocuk, x );
18:     return (T) ? T : agacta_ara( L->sagcocuk, x );
19: }
20:
21: int derinlik( agac_gost L ){
22:     int max( int a, int b ); // a ve b'den küçük olmayanı döndürür
23:     return (L) ? max( derinlik( L->solcocuk ),
24:         derinlik( L->sagcocuk ) ) + 1 : 0;
25: }
26:
27: int max( int a, int b ){
28:     return (a<b) ? b : a;
29: }
```


2009 Ara Sınav 2

1-i) Şekildeki örnek ağaç verildiğinde `AgirlikBul` yordamının $x=20$ için kaç kez işletileceğini ve $x=30$ için kaç kez işletileceğini yazınız.



```
1: int AgirlikBul(oge_gost L, int x) {
2:     if (!L)
3:         return 0;
4:     else if (L->veri == x)
5:         return agirlik(L);
6:
7:     if (AgirlikBul(L->solcocuk) > AgirlikBul(L->sagcocuk))
8:         AgirlikBul(L->solcocuk);
9:
10:    if (AgirlikBul(L->sagcocuk) > AgirlikBul(L->solcocuk))
11:        AgirlikBul(L->sagcocuk);
12:
13:    return 0;
14: }
15:
16: int agirlik(oge_gost L) {
17:     if (L)
18:         return MAX(agirlik(L->solcocuk) + agirlik(L->sagcocuk)) + 1;
19:     else
20:         return 0;
21: }
```

Öğrenci Çözümü (1-i):

$x=20$ için 14 kez, $x=30$ için 21 kez işletilir.

1-ii) `AgirlikBul` yordamının işletim süresini en kötü durum için veren özyineli ilişkiyi yazınız. Özyineli ilişkiyi çözünüz.

2-i) (Küme_Bul ve Küme_Birleşim yordamları verilmiş, 3 adet de küme görünümü çizilmiş.) İlk başta ata dizisinin içeriği nedir?

2-ii) Verilen karışık kümebul ve birleşim çağrılarının ardından ata dizisinin içeriği ne olur?
Not: i şıkkı hatalıysa ii şıkkı değerlendirilmez.

3-i) Merge sort yordamının verilen uyarlaması(versiyonu)yla eleman tipinde struct öğelerinden oluşan a dizisi, anahtar alanlarına göre sıralanmak istenmektedir. Mergesort yordamının özyineli çağrılarının yığıt çerçevesini çizim A'ya uygun olarak çiziniz. Eğer merge yordamı çağrılmışsa dönüş değerleri çizimdeki gibi belirtilmelidir.

```
1: typedef struct eleman {
2:     int anahtar;
3:     int bag;
4: };
5:
6: int mergesort(eleman a[], int L, int R) {
7:     int m;
8:     if (R <= L)
9:         return L;
10:
11:     m = (R + L) / 2;
12:     return merge(a, mergesort(a, m + 1, R), mergesort(a, L, m));
13: }
14:
15: int merge(eleman a[], int birinci, int ikinci) {
16:     int sonuc = 0;
17:     for (int i1 = birinci, i2 = ikinci; i1 && i2;)
18:         if (a[i1].anahtar <= a[i2].anahtar) {
19:             a[sonuc].bag = i1;
20:             sonuc = i1;
21:             i1 = a[i1].bag;
22:         } else {
23:             a[sonuc].bag = i2;
24:             sonuc = i2;
25:             i2 = a[i2].bag;
26:         }
27:
28:     if (i1 == 0)
29:         a[sonuc].bag = i2;
30:     else
31:         a[sonuc].bag = i1;
32:
33:     return a[0].bag;
34: }
```

3-ii) İşletim tablosunu merge çağrılarının numaralarını ve dönüş değerlerini de yazarak doldurunuz. Her adımda bag alanlarının değerlerini yazınız.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	Kaçıncı merge	Dönüş değeri	L, R, m
Anahtar		?	?	18	-5	27	11	?	83	30			
Bag	0	0	0	0	0	0	0	0	0	0	merge #1		

2009 Final

- 1) Kök-ortada (*in-order*) dolaşım sırası C, M, D, A, K, N, B, F, E, G olarak, kök-sonda (*post-order*) dolaşım sırası ise D, A, N, F, E, B, K, M, G, C olarak verilen ikili ağaç (*binary tree*) için ağaç görünümünü çiziniz. Tek bir ağaç görünümü vardır. (*unique*) (Yanıtınız çözüm yolunuzu da içermelidir.)

Öğrenci Çözümü (1):

Kök-ortada dolaşımında önce sol ağaç, sonra kök düğüm, en son ise sağ ağaç basılıyordu. Öyleyse kök olduğunu bildiğimiz bir düğümden sonra hiçbir düğüm basılmamışsa, bu düğümün sağ çocuğu yoktur; kökten önce düğüm basılmamışsa düğümün sağ çocuğu yoktur.

Kök-sonda dolaşımında önce sol ağaç, sonra sağ ağaç, en son ise kök düğüm basılıyordu. Öyleyse kök düğümler, kendi alt ağaçlarındaki düğümlerin hepsinden sonra basılır.

Öyleyse kök-sonda dolaşımında sondan başa doğru ilerleyeceğiz. Aldığımız düğüm her seferinde kök konumunda bulunacaktır. Sağ ve sol ağaçların durumuna karar vermek içinse kök-ortada dolaşımdan yardım alacağız.

Şimdi örneğimiz üzerinde düşünelim:

- 1- Kök-sonda dolaşımında en son basılan düğüm C'dir; o halde C ağacın kökü olmalıdır. Kök-ortada dolaşımında C'den önce düğüm basılmamış olduğuna göre C'nin sol çocuğu yoktur.
- 2- Şimdi C'nin sol çocuğu bulunmadığını biliyoruz. Öyleyse bundan sonraki düğümler sağ alt ağaçta yer alacaktır. Kök-sonda dolaşımında (kalan listede) en son G basıldığına göre G, sonraki düğümlerin oluşturacağı ağacın köküdür. Yani C'nin sağına eklenecek düğüm G'dir. Kök-ortada dolaşımında G'den sonra hiçbir düğüm basılmadığına göre G'nin sağ çocuğu yoktur.

kök-ortada (in-order) C M D A K N B F E G

kök-sonda (post-order) D A N F E B K M G C

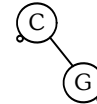
ADIM 1



kök-ortada (in-order) C M D A K N B F E G

kök-sonda (post-order) D A N F E B K M G C

ADIM 2



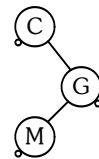
- 3- Kök-sonda dolaşımında sondan başa doğru ilerlemeye devam ediyoruz. Sıradaki eleman M'dir. Kök-ortada dolaşımında M'den önce basılan düğüm bulunmadığına göre M'nin sol ağacı boştur.

- 4- Sıradaki düğüm: K. Kök-ortada dolaşımında K'dan önce basılan iki düğüm görüyoruz: D, A. O halde D ve A düğümleri, K'nın sol alt ağacında yer almalıdır.

kök-ortada (in-order) C M D A K N B F E G

kök-sonda (post-order) D A N F E B K M G C

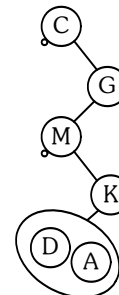
ADIM 3



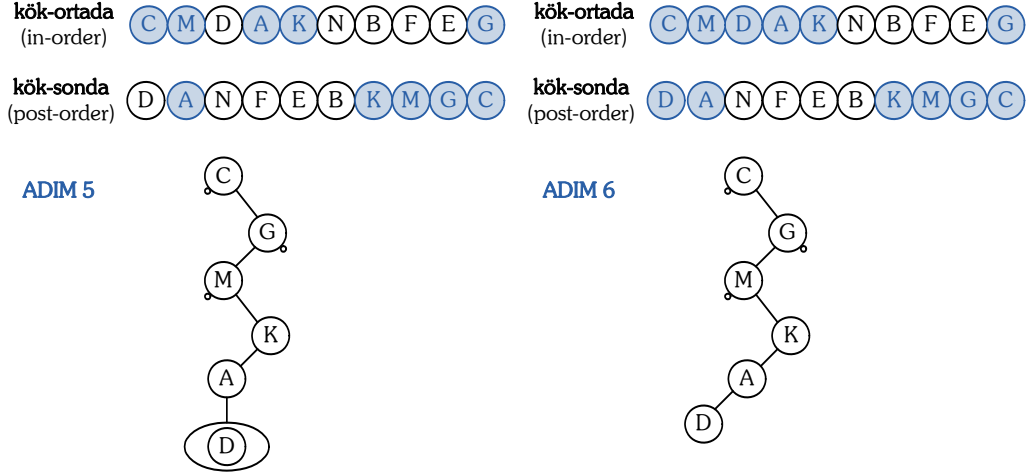
kök-ortada (in-order) C M D A K N B F E G

kök-sonda (post-order) D A N F E B K M G C

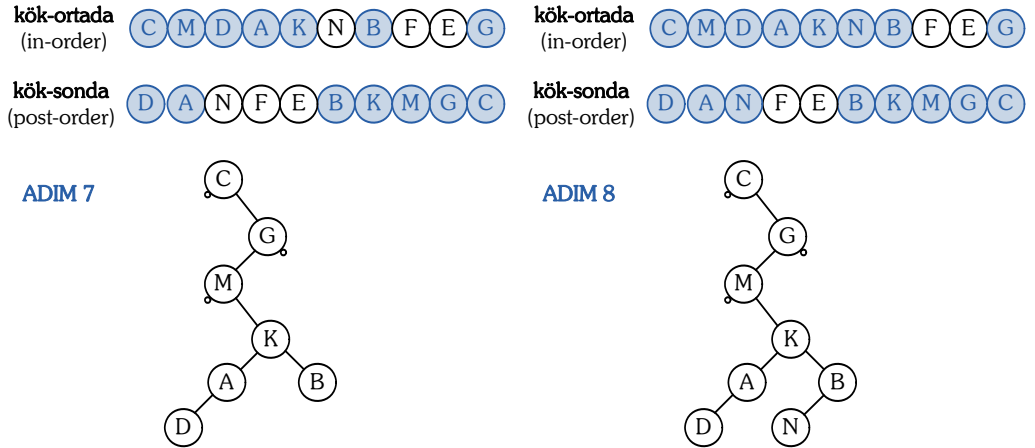
ADIM 4



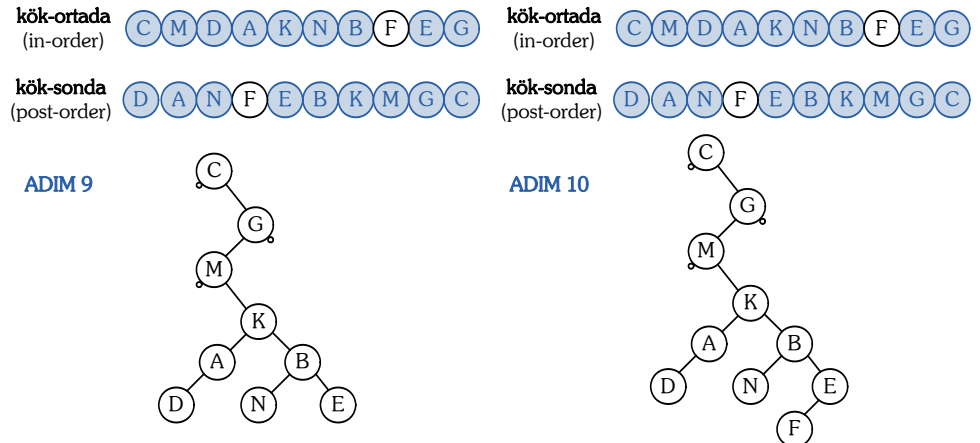
- 5- Şimdi D ve A düğümleri için yer arayalım. D ve A düğümleri bir alt ağaç oluşturacağından dolayı biri diğerinin atası olacaktır. Kök-sonda dolaşımında A, D'den sonra basıldığına göre A ata olmalıdır.
- 6- Peki D, A'nın sağ çocuğu mu olmalı, yoksa sol çocuğu mu olmalıdır? Kök-ortada dolaşımında D'nin daha önce basılabilmesi için D, sol çocuk olmalıdır.




- 7- K'nın sol alt ağacına D ve A'yı koyduk. Sıradaki elemanı K'nın sağ ağacına yerleştireceğiz.
- 8- Kök ortada dolaşımında B'den önce basılmış bir eleman var: N. Öyleyse N düğümü, B düğümünün sol çocuğu olacaktır.




- 9- Sıradaki düğüm: E. Kaldığımız konum olan B'nin sağına E'yi yerleştiriyoruz.
- 10- Son düğümü kök-ortada dolaşımında E'nin solunda yer aldığı E'nin sol çocuğu yapıyoruz.



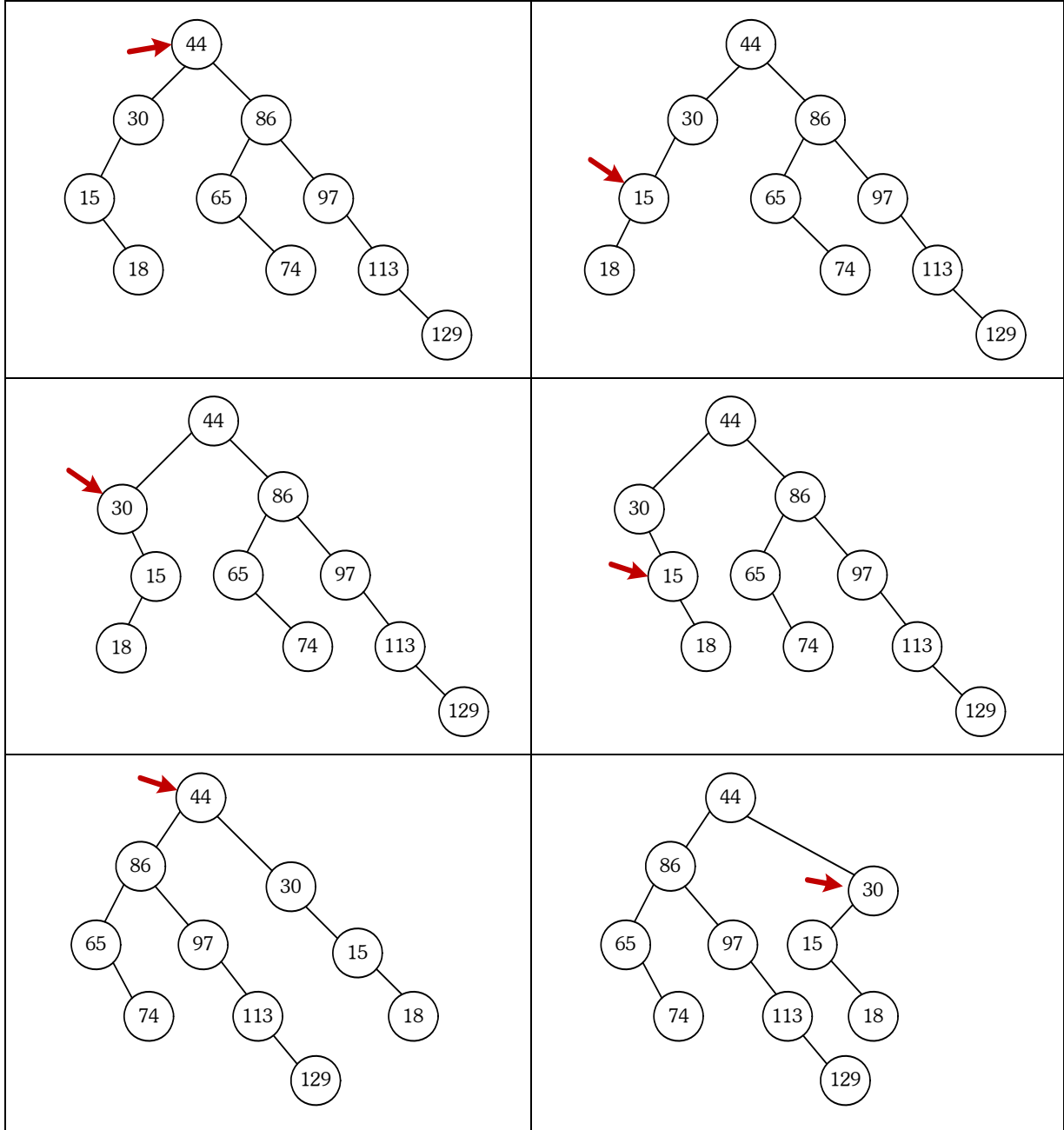
- 2) Sırasıyla {44, 30, 15, 18, 86, 97, 65, 113, 74, 129} verileri geldiğinde oluşan ikili arama ağacı (*binary search tree*) L göstergesi ile işaret edilmektedir. L ağacı için ağaç görünümünü çiziniz. L ağacı için yandaki yordamı işletiniz.  işareti bulunan satırdan her geçişte –eğer ağacın görünümü değişmişse– ağaç görünümünü tekrar çiziniz. Ağacın son halini de çiziniz ve yorumlayınız.

```

1: void yordam( agac_gost L ){
2:     agac_gost p;
3:     if(L){
4:         yordam( L->solcocuk );
5:         SWAP( L->solcocuk, L->sagcocuk, p );
6:         // -----  ----- //
6:         yordam( L->sagcocuk );
7:     }
8: }

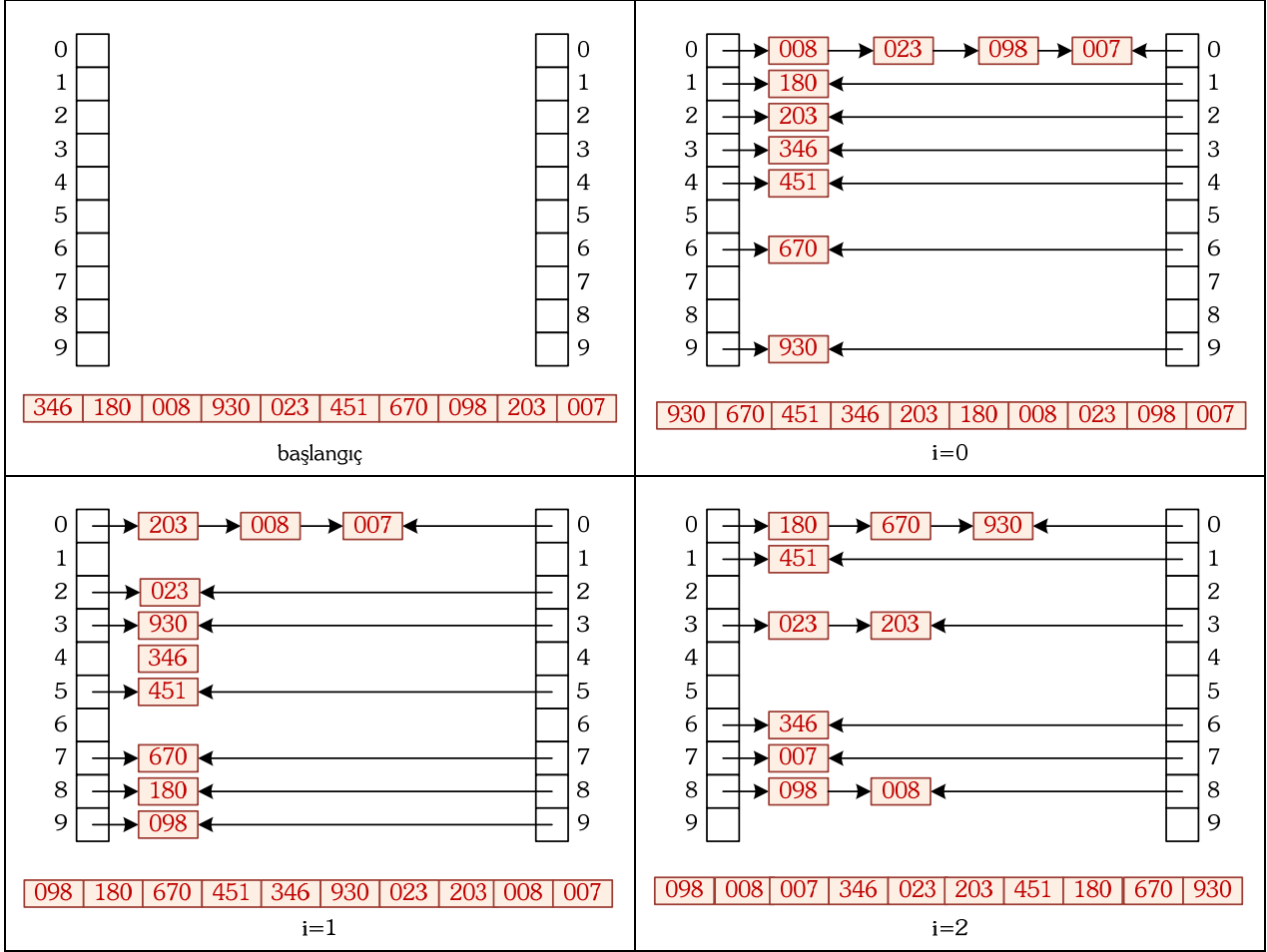
```

Öğrenci Çözümü (2):



SWAP komutu iki özyineli çağrının arasına değil de öncesine ya da sonrasına yazılmış olsaydı, bu yordam ağacın sağ ve sol çocuk ikililerini birbirleriyle yer değiştirdi. Ancak SWAP makrosundan önce sol çocuk konumunda bulunan alt ağaç, SWAP ile sağ çocuk haline gelmekte; sonra yordam aynı eleman için bir kez daha işletilmektedir.

Öğrenci Çözümü (4):



İlk değişirme ile Mert, birler basamağını en anlamlı basamak gibi kabul etmiş oldu. Dolayısıyla sayıların her biri basamakları ters sırada okunursa doğru sıralanmış olduğu görülür. Örneğin 098, 008, 007, 346 şeklinde elde edilen sıra 890, 800, 700, 643 olarak okunduğunda sıralı gitmektedir.

İkinci yapılan değişirme ise normalde küçükten büyüğe sıralanan verilerin bu kez büyükten küçüğe sıralanmasına sebep olmuştur.

LİNKLER

Veri yapıları listesi

[Link1](#)

[Link2](#)

[Link3](#)

[Link4](#)

[Link5](#)