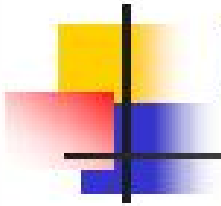
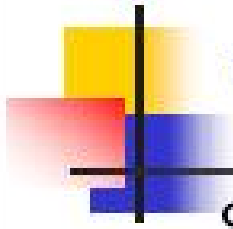
SIKIŞTIRMA ALGORİTMASI



Konular

- Veri Sıkıştırma
- Kayıplı ve Kayıpsız Sıkıştırma
- Sabit Genişlikli / Değişken Genişlikli Kodlama
- Huffman Algoritması (Greedy Algorithms)
  - Kodlama ağacının oluşturulması
  - Kodlamanın yapılması
  - Sıkıştırılmış bilginin tekrar elde edilmesi



Veri Sıkıştırma

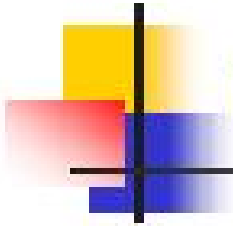
Sıkıştırma niçin kullanılır ?

1- Alan gereksinimini azaltmak için

- Hard disklerin boyutu büyümekle birlikte, yeni programların ve data dosyalarınında boyutu büyümektedir ve alan ihtiyacı artmaktadır
- 3.5" floppy diskler hala kullanılmaktadır ve alanları çok azdır

2- Zaman ve bant genişliği gereksinimini azaltmak için

- Birçok program ve dosya internetten download edilmektedir
- Birçok kişi düşük hızlı bağlantıyı kullanmaktadır
- Sıkıştırılmış dosyalar transfer süresini kısaltmakta ve birçok kişinin aynı server'ı kullanımına izin vermektedir



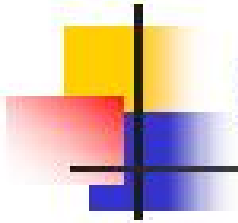
Kayıplı ve Kayıpsız Sıkıştırma

1- Kayıplı sıkıştırma

- Bilginin bir kısmı geri elde edilmez (MP3, JPEG, MPEG)
- Genellikle ses ve görüntü uygulamalarında kullanılır
- Çok büyük ses ve görüntü dosyalarında çok iyi sıkıştırma yapar ve kullanıcı kalitedeki azalmayı farketmez

2- Kayıpsız sıkıştırma

- Orijinal dosya tekrar tam olarak elde edilir (D(C(X)) = X, burada C sıkıştırılan dosyayı, D ise açılan dosyayı ifade eder)
- .txt, .exe, .doc, .xls, .java, .cpp vs. gibi dosyalarda kullanılır
- Ses ve görüntü dosyalarındada kullanılabilir ancak kayıplı sıkıştırma kadar yüksek sıkıştırma oranına sahip olmaz



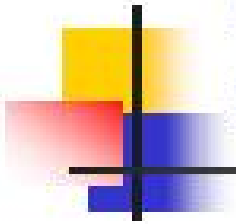
Kayıpsız Sıkıştırma

Kayıpsız sıkıştırma temel olarak 3 algoritma altında toplanır

- **Huffman** – her karakter için değişken genişlikli bir kelime kodu (codeword) kullanır
- LZ77 - "sliding" window kullanır ve bir grup karakteri bir zamanda sıkıştırır
- LZ78 / LZW – daha önce karşılaşılan işaretleri saklayan bir sözlük kullanır ve bir grup karakteri aynı anda sıkıştırır

Kayıpsız sıkıştırma uygulamaları

- unix compress, gif: LZW
- pkzip, zip, winzip, gzip: Huffman + LZ77



Kodlama Teorisi (Coding Theory)
Fixed Length ve Variable Length

Toplam 1.000.000.000 karakterden oluşan bilginin sadece 6 harften {a,b,c,d,e,f} oluştuğunu varsayalım. Karakterlerin kullanım sıklıkları ise aşağıdaki gibi olsun.

a	b	c	d	e	f	
.45	.13	.12	.16	.09	.05	Frekans
000	001	010	011	100	101	Fixed length
0	101	100	111	1101	1100	Variable length

Fixed length $3 \bullet 1.000.000.000 = 3.000.000.000$ bits $\approx$ **3GB**

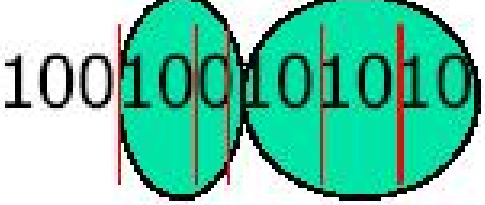
Variable length $(.45\bullet 1+.13\bullet 3+.12\bullet 3+.16\bullet 3+.09\bullet 3+.05\bullet 3)\bullet 1.000.000.000$
$= 2.240.000.000$ bits $\approx$ **2.24GB**

Karakterlerin Kodunu Çözme (Decode)

E	0
T	11
N	100
I	1010
S	1011

Prefix code

E	0
T	10
N	100
I	0111
S	1010



Belirsiz

Prefix code

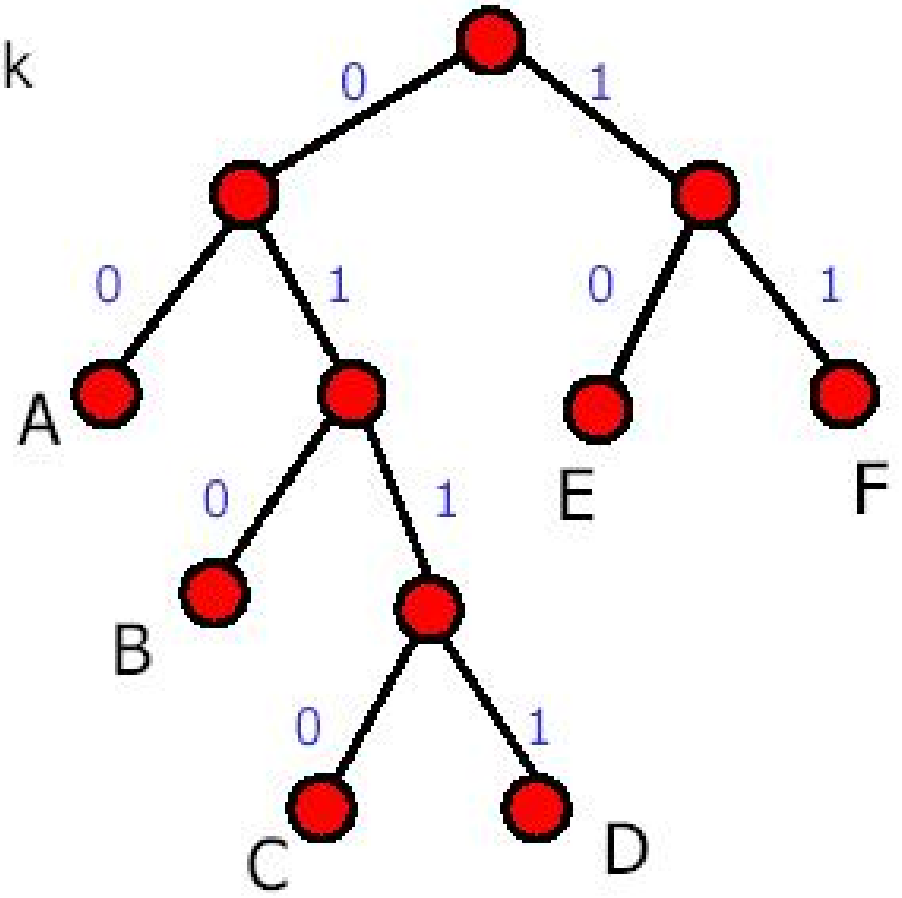
- Bir codeword başka bir codeword'ün prefix'i olamaz
- Kod çözülmesinde teklik olmalı

A	00	1	00
B	010	01	10
C	011	001	11
D	100	0001	0001
E	11	00001	11000
F	101	000001	101

Prefix code ve ikilik ağaç

- Prefix code'ların ikilik ağaçla gösterimi

A	00
B	010
C	0110
D	0111
E	10
F	11

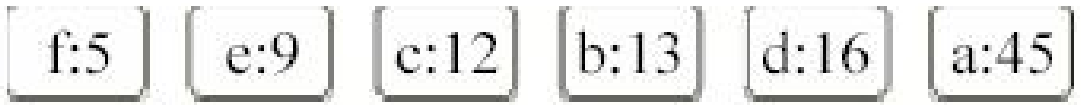


Kodlama için gereken boyut

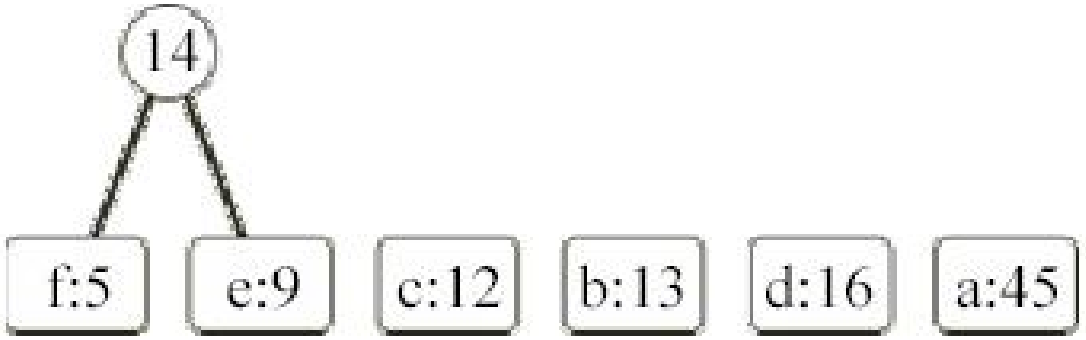
- $f(c)$ – c karakterinin dosya içindeki frekansını gösterirsin
- $d_T(c)$ – c karakterinin bulunduğu yaprağın seviyesini gösterirsin (kod boyutu)
- Kodlama için gereken dosyanın bit olarak boyutu $B(T)$ aşağıdaki gibi ifade edilir

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

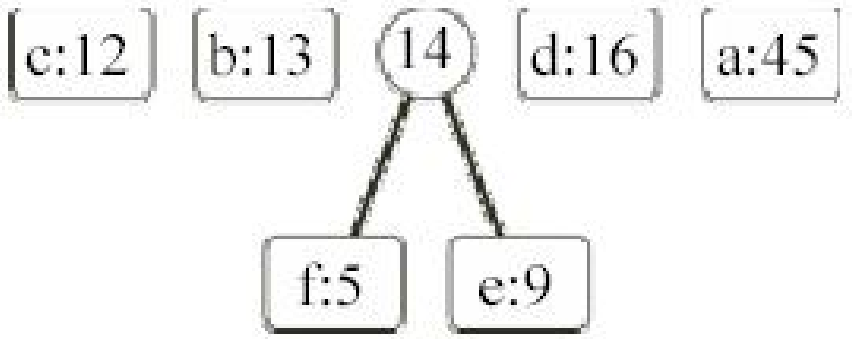
Kodlama ağacının oluşturulması



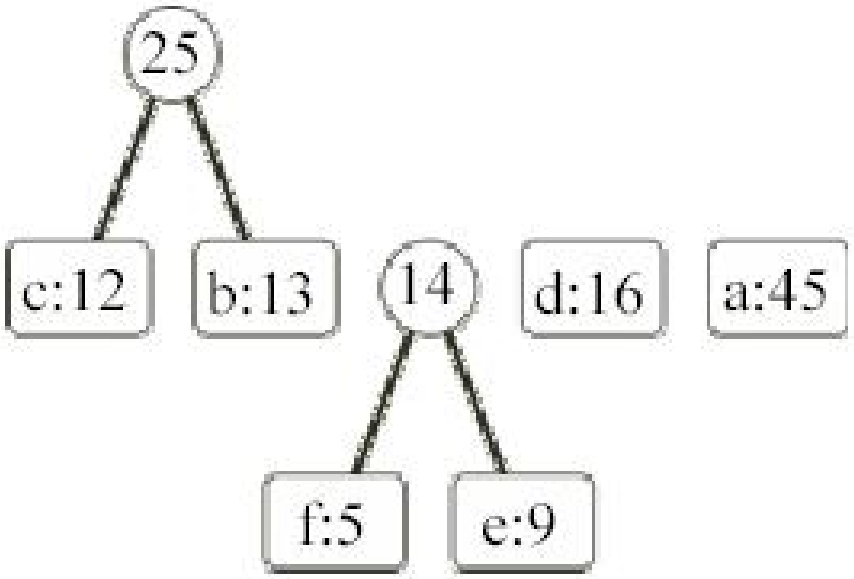
Kodlama ağacının oluşturulması



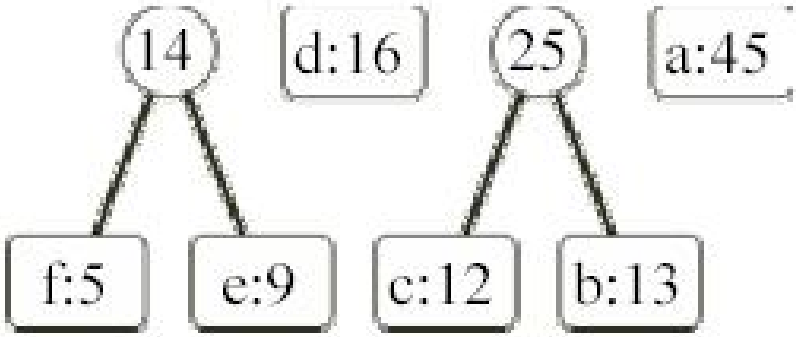
Kodlama ağacının oluşturulması



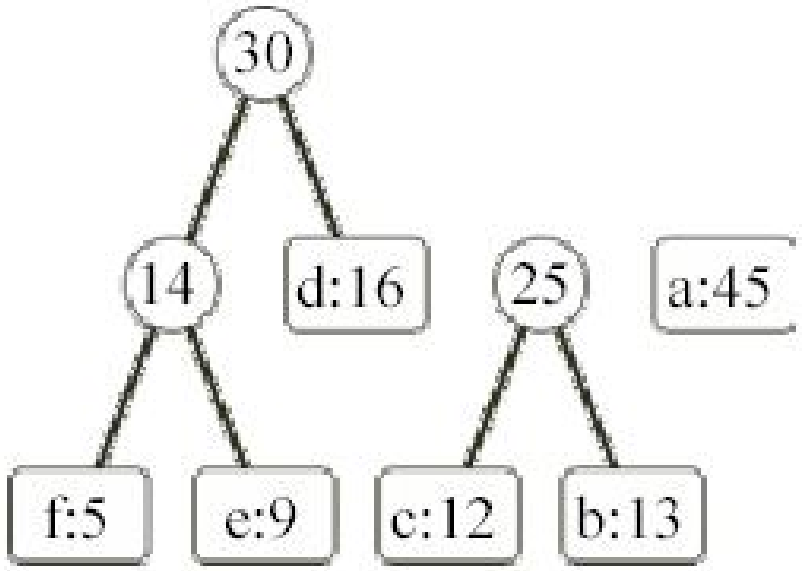
Kodlama ağacının oluşturulması



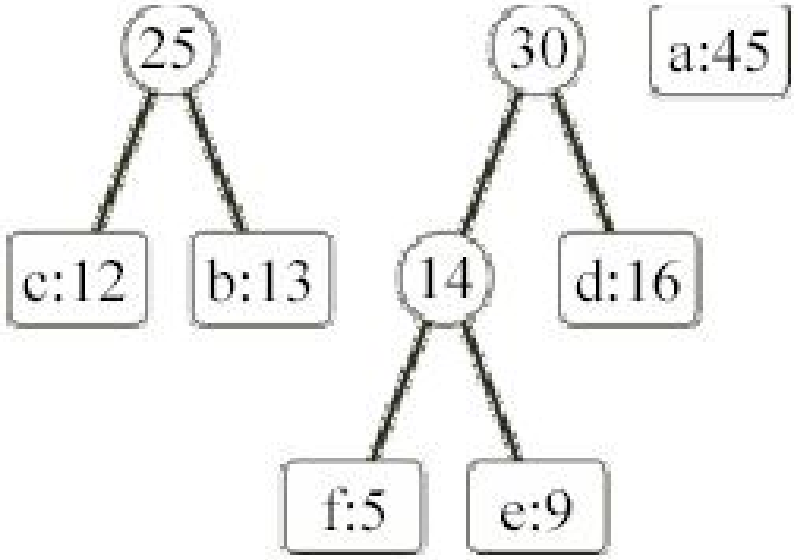
Kodlama ağacının oluşturulması



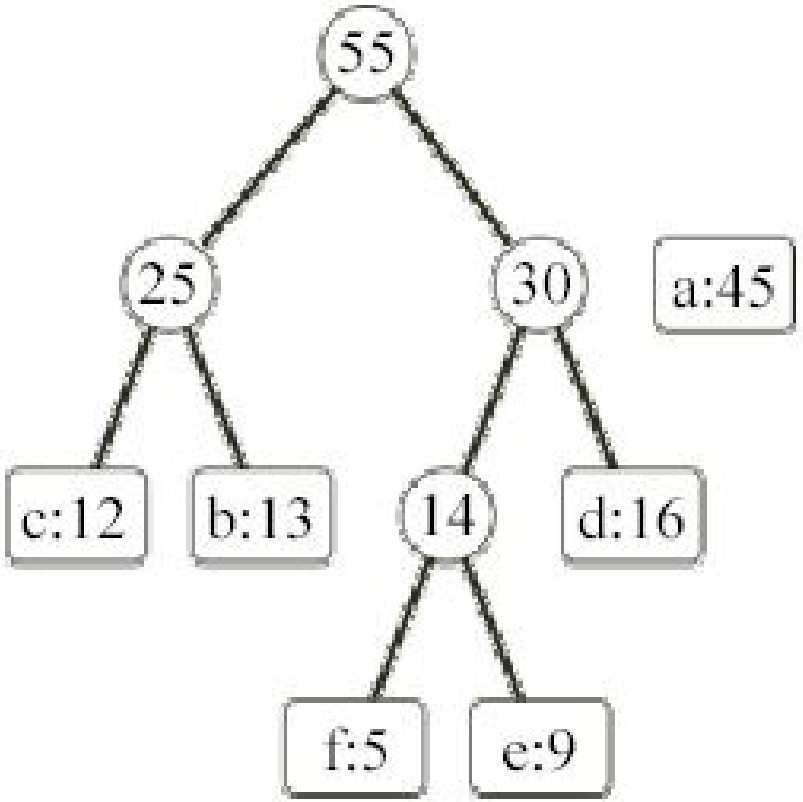
Kodlama ağacının oluşturulması



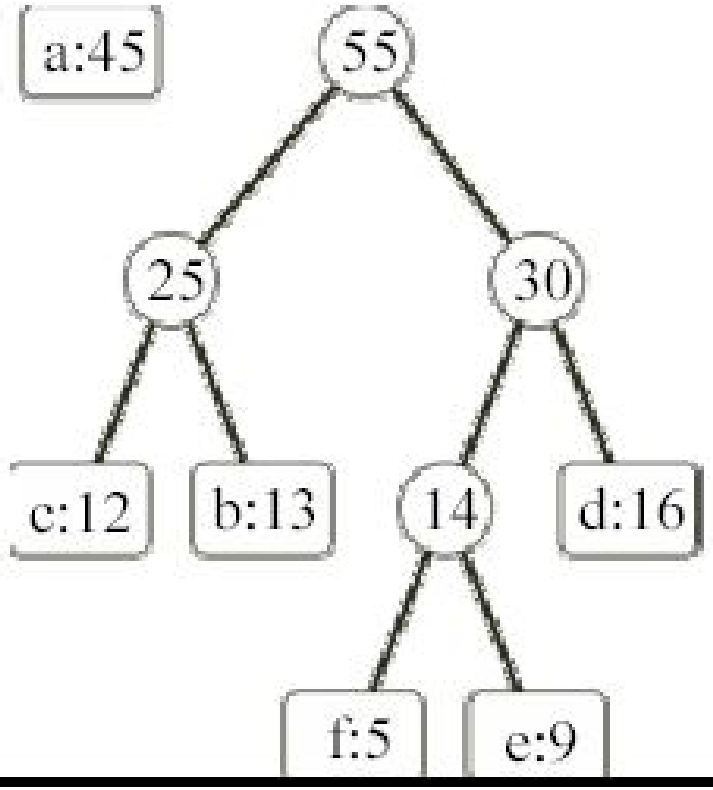
Kodlama ağacının oluşturulması



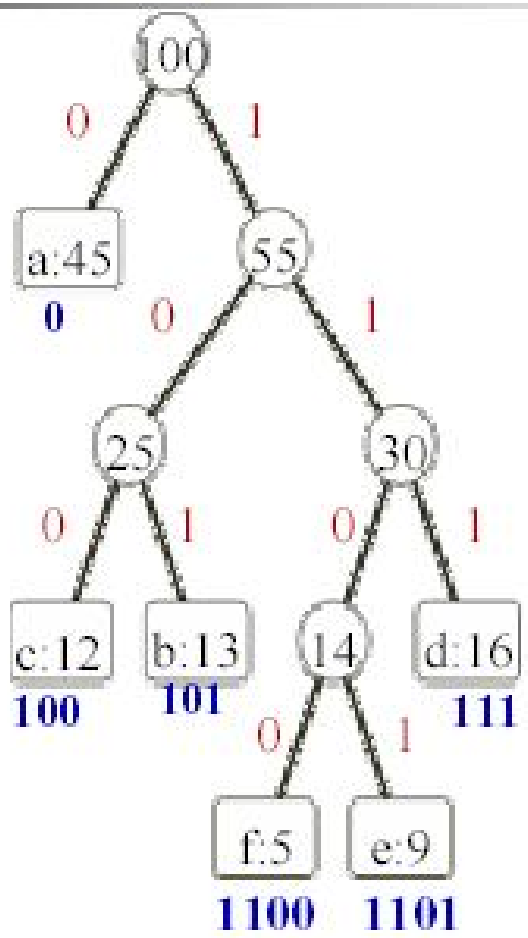
Kodlama ağacının oluşturulması



Kodlama ağacının oluşturulması



Kodlama ağacının oluşturulması



Huffman algoritması

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

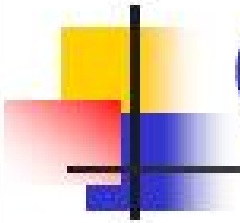
Farklı karakter sayısı

Min-Priority Queue

En düşük frekanslı iki node

İki node'un toplamı yeni node

Yeni node kuyruktaki yerine ekleniyor



Çalışma süresi

- Uygun veri yapısı olarak min-heap kullanılabilir
- Heap oluşturma O(*lg n)* ve for döngüsünde *n-1* adet işlem yapılır
- Toplam çalışma süresi O(*n lg n*) olur

DAVİD HUFFMAN ALGORİTMASI

- Sayısal haberleşme tekniklerinin önemli ölçüde arttığı günümüzde, sayısal verilen iletilmesi ve saklanması bir hayli önem kazanmıştır. Sayısal veriler çeşitli saklayıcılarda saklanırken hedef daima minimum alanda maksimum veriyi saklamadır. Veriler çeşitli yöntemlerle sıkıştırılarak kapladığı alandan ve iletim zamanından tasarruf edilir. Sayısal iletişim(digital communication) kuramında veriler çok çeşitli yöntemlerle sıkıştırılabilir. Bu yöntemlerden en çok bilineni **David Huffman** tarafından öne sürülmüştür. Bu yazıda bu teknik "Huffman algoritması" olarak adlandırılacaktır. Bu yazıda Huffman Algoritması detaylı olarak açıklandıktan sonra bu algoritmanın C# dili ile ne şekilde uygulanacağı gösterilecektir.

  Sıkıştırma algoritmaları temel olarak iki kategoride incelenir. Bunlar, kayıplı ve kayıpsız sıkıştırma algoritmalarıdır. Kayıplı algoritmalarda sıkıştırılan veriden orjinal veri elde edilemezken kayıpsız sıkıştırma algoritmalarında sıkıştırılmış veriden orjinal veri elde edilebilir. Kayıplı sıkıştırma tekniklerine verilebilecek en güzel örnekler MPEG ve JPEG gibi standartlarda kullanılan sıkıştırmalardır. Bu tekniklerde sıkıştırma oranı artırıldığında orjinal veride bozulmalar ve kayıplar görülür. Örneğin sıkıştırılmış resim formatı olan JPEG dosyalarının kaliteli yada az kaliteli olmasının nedeni sıkıştırma katsayısıdır. Yani benzer iki resim dosyasından daha az disk alanı kaplayan daha kötü kalitededir deriz. Kayıpsız veri sıkıştırmada durum çok farklıdır. Bu tekniklerde önemli olan orjinal verilerin aynen sıkıştırılmış veriden elde edilmesidir. Bu teknikler daha çok metin tabanlı verilen sıkıştırılmasında kullanılır. Bir metin dosyasını sıkıştırdıktan sonra metindeki bazı cümlelerin kaybolması istenmediği için metin sıkıştırmada bu yöntemler kullanılır

- Bu yazının konusu olan Huffman sıkıştırma algoritması kayıpsız bir veri sıkıştırma tekniğini içerir. Bu yüzden bu yöntemin en elverişli olduğu veriler metin tabanlı verilerdir. Bu yazıda verilecek örnek programdaki hedef metin tabanlı verilerin sıkıştırılması olacaktır.

  Huffman algoritması, bir veri kümesinde daha çok rastlanan sembolü daha düşük uzunluktaki kodla, daha az rastlanan sembolleri daha yüksek uzunluktaki kodlarla temsil etme mantığı üzerine kurulmuştur. Bir örnekten yola çıkacak olursak : Bilgisayar sistemlerinde her bir karakter 1 byte yani 8 bit uzunluğunda yer kaplar. Yani 10 karakterden oluşan bir dosya 10 byte büyüklüğündedir. Çünkü her bir karakter 1 byte büyüklüğündedir. Örneğimizdeki 10 karakterlik veri kümesi "aaaaaaaccs" olsun. "a" karakteri çok fazla sayıda olmasına rağmen "s" karakteri tektir. Eğer bütün karakterleri 8 bit değilde veri kümesindeki sıklıklarına göre kodlarsak veriyi sembolize etmek için gereken bitlerin sayısı daha az olacaktır. Söz gelimi "a" karakteri için "0" kodunu "s" karakteri için "10" kodunu, "c" karakteri için "11" kodunu kullanabiliriz. Bu durumda 10 karakterlik verimizi temsil etmek için (a kodundaki bit sayısı)*(verideki a sayısı) + (c kodundaki bit sayısı)*(verideki c sayısı) + (s kodundaki bit sayısı)*(verideki s sayısı) = 1*7 + 2*2 + 2*1 = 12 bit gerekecektir. Halbuki bütün karakterleri 8 bit ile temsil etseydik 8*10 = 80 bite ihtiyacımız olacaktı. Dolayısıyla %80 'in üzerinde bir sıkıştırma oranı elde etmiş olduk. Burada dikkat edilmesi gereken nokta şudur : Veri kümesindeki sembol sayısına ve sembollerin tekrarlanma sıklıklarına bağlı olarak Huffman sıkıştırma algoritması %10 ile %90 arasında bir sıkıştırma oranı sağlayabilir. Örneğin içinde 2000 tane "a" karakteri ve 10 tane "e" karakteri olan bir veri kümesi Huffman tekniği ile sıkıştırılırsa %90'lara varan bir sıkıştırma oranı elde edilir.

- ***Huffman tekniğinde semboller(karakterler) ASCII'de olduğu gibi sabit uzunluktaki kodlarla kodlanmazlar. Her bir sembol değişken sayıda uzunluktaki kod ile kodlanır.***

  Bir veri kümesini Huffman tekniği ile sıkıştırabilmek için veri kümesinde bulunan her bir sembolün ne sıklıkta tekrarlandığını bilmemiz gerekir. Örneğin bir metin dosyasını sıkıştırıyorsak her bir karakterin metin içerisinde kaç adet geçtiğini bilmemiz gerekiyor. Her bir sembolün ne sıklıkta tekrarlandığını gösteren tablo **frekans tablosu** olarak adlandırılmaktadır. Dolayısıyla sıkıştırma işlemine geçmeden önce frekans tablosunu çıkarmamız gerekmektedir. Bu yönteme Statik Huffman tekniği de denilmektedir. Diğer bir teknik olan Dinamik Huffman tekniğinde sıkıştırma yapmak için frekans tablosuna önceden ihtiyaç duyulmaz. Frekans tablosu her bir sembolle karşılaştıkça dinamik olarak oluşturulur. Dinamik Huffman tekniği daha çok haberleşme kanalları gibi hangi verinin geleceği önceden belli olmayan sistemlerde kullanılmaktadır. Bilgisayar sistemlerindeki dosyaları sıkıştırmak için statik huffman metodu yeterlidir. Nitekim bir dosyayı baştan sona tarayarak herbir sembolün hangi sıklıkla yer aldığını tespit edip frekans tablosunu elde etmemiz çok basit bir işlemdir.

  Huffman sıkıştırma tekniğinde frekans tablosunu elde etmek için statik ve dinamik yaklaşımlarının olduğunu söyledik. Eğer statik yöntem seçilmişse iki yaklaşım daha vardır. Birinci yaklaşım, metin dosyasının diline göre sabit bir frekans tablosunu kullanmaktır. Örneğin Türkçe bir metin dosyasında "a" ve "e" harflerine çok sık rastlanırken "ğ" harfine çok az rastlanır. Dolayısıyla "ğ" harfi daha fazla bitle "a" ve "e" harfi daha az bitle kodlanır. Frekans tablosunu elde etmek için kullanılan diğer bir yötem ise metni baştan sona tarayarak her bir karakterin frekansını bulmaktır. Sizde takdir edersinizki ikinci yöntem daha gerçekçi bir çözüm üretmekle beraber metin dosyasının dilinden bağımsız bir çözüm üretmesi ile de ön plandadır. Bu yöntemin dezavantajı ise sıkıştırılan verilerde geçen sembollerin frekansının da bir şekilde saklanma zorunluluğunun olmasıdır. Sıkıştırılan dosyada her bir sembolün frekansıda saklanmalıdır. Bu da küçük boyutlu dosyalarda sıkıştırma yerine genişletme etkisi yaratabilir. Ancak bu durum Huffman yönteminin kullanılabililiğini zedelemez. Nitekim küçük boyutlu dosyaların sıkıştırılmaya pek fazla ihtiyacı yoktur zaten.

- Frekans tablosunu metin dosyasını kullanarak elde ettikten sonra yapmamız gereken **"Huffman Ağacını"** oluşturmaktır. Huffman ağacı hangi karakterin hangi bitlerle temsil edileceğini(kodlanacağını) belirlememize yarar. Birazdan örnek bir metin üzerinden "Huffman Ağacını" teorik olarak oluşturup algoritmanın derinliklerine ineceğiz. Bu örneği iyi bir şekilde incelediğinizde Huffman algoritmasının aslında çok basit bir temel üzerine kurulduğunu göreceksiniz.

- **Huffman Ağacının Oluşturulması**

- Bir huffman ağacı aşağıdaki adımlar izlenerek oluşturulabilir.

  Bu örnekte aşağıdaki frekans tablosu kullanılacaktır.

  Sembol(Karakter)Sembol Frekansı a50b35k20m10d8ğ4Bu tablodan çıkarmamız gereken şudur : Elimizde öyle bir metin dosyası varki "a" karakteri 50 defa, "b" karakteri 35 defa .... "ğ" karakteri 2 defa geçiyor. Amacımız ise her bir karakteri hangi bit dizileriyle kodlayacağımızı bulmak

- **1** - Öncelikle "Huffman Ağacını" ndaki en son düğümleri(dal) oluşturacak bütün semboller frekanslarına göre aşağıdaki gibi küçükten büyüğe doğru sıralanırlar.

- **2** - En küçük frekansa sahip olan iki sembolün frekansları toplanarak yeni bir düğüm oluşturulur. Ve oluşturulan bu yeni düğüm diğer varolan düğümler arasında uygun yere yerleştirilir. Bu yerleştirme frekans bakımından küçüklük ve büyüklüğe göredir. Örneğin yukarıdaki şekilde "ğ" ve "d" sembolleri toplanarak "12" frakansında yeni bir "ğd" düğümü elde edilir. "12" frekanslı bir sembol şekilde "m" ve "k" sembolleri arasında yerleştirilir. "ğ" ve "d" düğümleri ise yeni oluşturulan düğümün dalları şeklinde kalır. Yeni dizimiz aşağıdaki şekilde olacaktır.

- **3** - 2.adımdaki işlem tekrarlanarak en küçük frekanslı iki düğüm tekrar toplanır ve yeni bir düğüm oluşturulur. Bu yeni düğümün frekansı 22 olacağı için "k" ve "b" düğümleri arasına yerleşecektir. Yeni dizimiz aşağıdaki şekilde olacaktır.

4.adımdaki işlem tekrarlanarak en küçük frekanslı iki düğüm tekrar toplanır ve yeni bir düğüm oluşturulur. Bu yeni düğümün frekansı 42 olacağı için "b" ve "a" düğümleri arasına yerleşecektir. Yeni dizimiz aşağıdaki şekilde olacaktır. Dikkat ederseniz her dalın en ucunda sembollerimiz bulunmaktadır. Dalların ucundaki düğümlere özel olarak **yaprak(leaf)** denilmektedir. Sona yaklaştıkça Bilgisayar bilimlerinde önemli bir veri yapısı olan Tree(ağaç) veri yapısına yaklaştığımızı görüyoruz..

5 - 2.adımdaki işlem tekrarlanarak en küçük frekanslı iki düğüm tekrar toplanır ve yeni bir düğüm oluşturulur. Bu yeni düğümün frekansı 77 olacağı için "a" düğümünden sonra yerleşecektir. Yeni dizimiz aşağıdaki şekilde olacaktır. Dikkat ederseniz her bir düğümün frekansı o düğümün sağ ve sol düğümlerinin frekanslarının toplamına eşit olmaktadır. Aynı durum düğüm sembolleri içinde geçerlidir.

6 - 2.adımdaki işlem en tepede tek bir düğüm kalana kadar tekrar edilir. En son kalan düğüm Huffman ağacının kök düğümü(root node) olarak adlandırılır. Son düğümün frekansı 127 olacaktır. Böylece huffman ağacının son hali aşağıdaki gibi olacaktır.

Not : Dikkat ederseniz Huffman ağacının son hali simetrik bir yapıda çıktı. Yani yaprak düğümler hep sol tarafta çıktı. Bu tamamen seçtiğimiz sembol frekanslarına bağlı olarak rastlantısal bir sonuçtur. Bu rastlantının oluşması oluşturduğumuz her bir yeni düğümün yeni dizide ikinci sıraya yerleşmesinden kaynaklanmaktadır. Sembol frekanslarında değişiklik yaparak ağacın şeklindeki değişiklikleri kendinizde görebilirsiniz.

7 - Huffman ağacının son halini oluşturduğumuza göre her bir sembolün yeni kodunu oluşturmaya geçebiliriz. Sembol kodlarını oluşturuken Huffman ağacının en tepesindeki kök düğümden başlanır. Kök düğümün sağ ve sol düğümlerine giden dala sırasıyla "0" ve "1" kodları verilir. Sırası ters yönde de olabilir. Bu tamamen seçime bağlıdır. Ancak ilk seçtiğiniz sırayı bir sonraki seçimlerde korumanız gerekmektedir. Bu durumda "a" düğümüne gelen dal "0", "bkmğd" düğümüne gelen dal "1" olarak seçilir. Bu işlem ağaçtaki tüm dallar için yapılır. Dalların kodlarla işaretlenmiş hali aşağıdaki gibi olacaktır.

- **8** - Sıra geldi her bir sembolün hangi bit dizisiyle kodlanacağını bulmaya. Her bir sembol dalların ucunda bulunduğu için ilgili yaprağa gelene kadar dallardaki kodlar birleştirilip sembollerin kodları oluşturulur. Örneğin "a" karakterine gelene kadar yalnızca "0" dizisi ile karşılaşırız. "b" karakterine gelene kadar önce "1" dizisine sonra "0" dizisi ile karşılaşırız. Dolayısıyla "b" karakterinin yeni kodu "10" olacaktır. Bu şekilde bütün karakterlerin sembol kodları çıkarılır. Karakterlerin sembol kodları aşağıda bir tablo halinde gösterilmiştir.
- FrekansSembol(Karakter)Bit SayısıHuffman Kodu
50a1035b21020k311010m411108d5111114ğ511110S
- Sıkıştırılmış veride artık ASCII kodları yerine Huffman kodları kullanılacaktır. Dikkat ederseniz hiçbir Huffman kodu bir diğer Huffman kodunun ön eki durumunda değildir. Örneğin ön eki "110" olan hiç bir Huffman kodu mevcut değildir. Aynı şekilde ön eki "0" olan hiç bir Huffman kodu yoktur. Bu Huffman kodları ile kodlanmış herhangi bir veri dizisinin **"tek çözülebilir bir kod"** olduğunu göstermektedir. Yani sıkıştırılmış veriden orjinal verinin dışında başka bir veri elde etme ihtimali sıfırdır. (Tabi kodlamanın doğru yapıldığını varsayıyoruz.)

  Şimdi örneğimizdeki gibi bir frekans tablosuna sahip olan metnin Huffman algoritması ile ne oranda sıkışacağını bulalım :

  Sıkıştırma öncesi gereken bit sayısını bulacak olursak : Her bir karakter eşit uzunlukta yani 8 bit ile temsil edildiğinden toplam karakter sayısı olan (50+35+20+10+8+4) = 127 ile 8 sayısını çarpmamız lazım. Orjinal veriyi sıkıştırmadan saklarsak **127*8 = 1016** bit gerekmektedir.

  Huffman algoritmasını kullanarak sıkıştırma yaparsak kaç bitlik bilgiye ihtiyaç duyacağımızı hesaplayalım : 50 adet "a" karakteri için 50 bit, 35 adet "b" karakteri için 70 bit, 20 adet "k" karakteri için 60 bit....4 adet "ğ" karakteri için 20 bite ihtiyaç duyarız. (yukarıdaki tabloya bakınız.) Sonuç olarak gereken toplam bit sayısı = 50*1 + 35*2 + 20*3 + 10*4 + 8*5 + 4*5 = 50 + 70 + 60 + 40 + 40 + 20 = 280 bit olacaktır.

  Sonuç : 1016 bitlik ihtiyacımızı 280 bite indirdik. Yani yaklaşık olarak %72 gibi bir sıkıştırma gerçekleştirmiş olduk. Gerçek bir sistemde sembol frekanslarınıda saklamak gerektiği için sıkıştırma oranı %72'ten biraz daha az olacaktır. Bu fark genelde sıkıştırılan veriye göre çok çok küçük olduğu için ihmal edilebilir.

- **Huffman Kodunun Çözülmesi**
- Örnekte verilen frekans tablosuna sahip bir metin içerisindeki "aabkdğmma" veri kümesinin sıkıştırılmış hali her karakter ile karakterin kodu yer değiştirilerek aşağıdaki gibi elde edilir.

  a a b k d ğ m m a
  0 0 10 110 11111 11110 1110 1110 0 --> 001011011111111110111011100

  Eğer elimizde frekans tablosu ve sıkıştırılmış veri dizisi varsa işlemlerin tersini yaparak orjinal veriyi elde edebiliriz. Şöyleki; sıkıştırılmış verinin ilk biti alnır. Eğer alınan bit bir kod sözcüğüne denk geliyorsa, ilgili kod sözcüğüne denk düşen karakter yerine koyulur, eğer alınan bit bir kod sözcüğü değilse sonraki bit ile birlikte ele alınır ve yeni dizinin bir kod sözcüğü olup olmadığına bakılır. Bu işlem dizinin sonuna kadar yapılır ve huffman kodu çözülür. Huffman kodları tek çözülebilir kod olduğu için bir kod dizisinden farklı semboller elde etmek olanaksızdır. Yani bir huffman kodu ancak ve ancak bir şekilde çözülebilir. Bu da aslında yazının başında belirtilen kayıpsız sıkıştırmanın bir sonucudur.

  C# ile Huffman Algoritmasının Gerçekleştirilmesi
- Huffman algoritması bilgisayar bilimlerinde genellikle metin dosyalarının sıkıştırılmasında kullanılır. Bu yazıda örnek bir program ile Huffman algoritmasının ne şekilde uygulanacabileceğini de göreceğiz. Dil olarak tabiki C#'ı kullanılacaktır.

- Huffman algoritmasının uygulanmasındaki ilk adım frekans tablosunun bulunmasıdır. Bu yüzden bir dosyayı sıkıştırmadan önce dosyadaki her bir karakterin ne sıklıkla yer aldığını bulmak gerekir. Örnek programda sembol frekanslarını bellekte tutmak için System.Collections isim alanıdan bulunan Hashtable koleksiyon yapısı kullanılmıştır. Örneğin "c" karakterinin frekansı 47 ise,

  CharFrequencies["c"] = 47

  şeklinde sembolize edilir. Örnek programda frekans tablosu aşağıdaki metot ile elde edilebilir. Sıkıştırılacak dosya baştan sona taranarak frekanslar hashtable nesnesine yerleştirilir.

- Frekans tablosunu yukarıdaki metot yardımıyla oluşturduktan sonra huffman algoritmasının en önemli adımı olan huffman ağacının oluşturulmasına sıra geldi. Huffman ağacının oluşturulması en önemli ve en kritik noktadır. Huffman ağacı oluşturulurken Tree(ağaç) veri yapısından faydalanılmıştır. Ağaçtaki her bir düğümü temsil etmek için bir sınıf yazılmıştır. Huffman ağacındaki düğümleri temsil etmek için kullanılabilecek en basit düğüm sınıfının yapısı aşağıdaki gibidir.

```
private Hashtable CharFrequencies =
new Hashtable();
```

```
private void MakeCharFrequencies()
{
FileStream fs = File.Open(file,FileMode.Open,FileAccess.Read);
StreamReader sr = new
StreamReader(fs,System.Text.Encoding.ASCII);
int iChar;
char ch;
while((iChar = sr.Read()) != -1)
{
ch = (char)iChar;
if(!CharFrequencies.ContainsKey(ch.ToString()))
{
CharFrequencies.Add(ch.ToString(),1);
}
else
{
int oldFreq = (int)CharFrequencies[ch.ToString()];
int newFreq;
newFreq = oldFreq + 1;
CharFrequencies[ch.ToString()] = newFreq;
}
}
sr.Close();
fs.Close();
sr = null;
fs = null;
}
```

- Frekans tablosunu yukarıdaki metot yardımıyla oluşturduktan sonra huffman algoritmasının en önemli adımı olan huffman ağacının oluşturulmasına sıra geldi. Huffman ağacının oluşturulması en önemli ve en kritik noktadır. Huffman ağacı oluşturulurken Tree(ağaç) veri yapısından faydalanılmıştır. Ağaçtaki her bir düğümü temsil etmek için bir sınıf yazılmıştır. Huffman ağacındaki düğümleri temsil etmek için kullanılabilecek en basit düğüm sınıfının yapısı aşağıdaki gibidir.

Yukarıdaki düğüm yapısı algoritmayı uygulamak için gereken en temel düğüm yapısıdır. Bu yapı istenildiği gibi genişletilebilir. Önemli olan minimum gerekliliği sağlamaktır. Düğüm sınıfınaki SagDüğüm ve SolDüğüm özellikleri de düğüm türündendir. Başlangıçtaki düğümler için bu değerler null dır. Her bir yeni düğümm oluşturulduğunda yeni düğümün sağ ve sol düğümleri oluşur. Ancak unutmamak gerekir ki sadece sembollerimizi oluşturan yaprak düğümlerde bu özellikler null değerdedir.

Örnek uygulamadak **HuffmanNode** sınıfında ayrıca ilgili düğümün yaprak olup olmadığını gösteren IsLeaf ve ilgili düğümün ana(parent) düğümünü gösteren HuffmanNode türünden ParentNode özellikleri bulunmaktadır

HAZIRLAYANLAR

MURAT BAKIR 360108019

ERDİNÇ YILDIRIM 360108024