25/02/2013

ALGORİTMALAR

Raziye AYDIN

✓ Analiz

Big-O

✓ Anagram problemi → 4 farklı yaklaşımla çözmeye çalışacağız. Amacımız tamamen çözmek değil.

→ Arama algoritmaları

→ Hash (karpı, sözlük)

→ Sıralama algoritmaları

→ Quick

✓ Ağaç Veri Yapısı (Tree)

Gerçek hayattaki ağaç ile mantıkları aynı.

→ İkil ağaç (Binary'den geliyor)

→ İkil Arama (Binary Search)

✓ Yığın (Heap)

→ Çizge (Graph)

→ BFS / DFS Algoritmaları

1) Hedefler

Problem
↓

Gösterimler

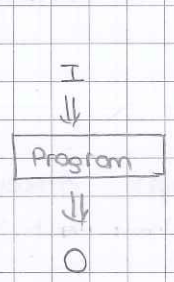
Kaba kod
Akış Diyagramı
Platform (win / Linux / mac)
Dil (Py / C / ...) (Win7 / XP / ... Ubuntu / ...)
Versiyon (Py 3)
İşlemci (...)
Bellek (...)

Kodlamaya başladık
↓
Program

GIPTA

TODAY

Samsun'dan İstanbul'a gidecek problemi en kısa sürede çözen program bu dersin konusu olur



En Kısa

Pentinium → 1sn
i7 → 1/10 sn
⋮ → 1/100 sn

Bunlar ne kadar adil bir yarış içinde Bunlardan hangisinin daha hızlı olduğunu, nasıl söyleyeceğimizi konuşacağız

* Algoritmaları kıyaslayarak hangisinin en kısa sürede olduğunu çözme işi algoritma analizidir.

Algoritmanın içinde bilgisayarı yoran işlemler vardır. Programda bir döngü n kez ise n döngüde n^2 kez yapar. 1.de 1 kez, 2.de n kez. Burada en hızlı olan 1 kere de olsa da platform ve dil gelince n^2 daha hızlı olabilir. Algoritmanın başarısının yüksekliği platformdan bağımsız olarak değerlendirilir.

→ Ardıl Arama: "Burak" ismini bulana kadar bütün isimleri aramaya denir.

→ İkili Arama: "Burak" ismini bulmak için;

sınıfı adlara göre sırala ve "Burak", B'ler arasında aramaya denir.

→ B'ler kendi de alfabetik sıralama olabilir.

① 2) Algoritma Analizi

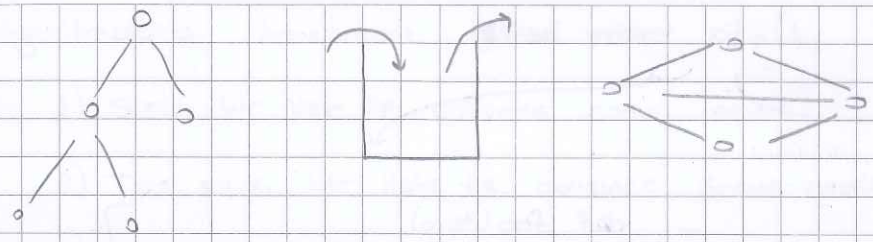
Algoritma: Problemi çözüme götüren işlem yumağıdır.

Bilgisayar programı olarak gerçeklenmesi uygun olan prob. çözme yöntemidir.

Veri Yapıları: Algoritmayı çözüme götürürken hesaplama yapılır. Yapılan bu hesaplamada veriyi organize etmeye denir.

Eğer doğru veri yapısı kullanılırsa algoritmayı anlamak ve çözmek kolaylaşır. Bu yüzden çok fazla veri yapısı vardır.

* Algoritmayı anlamak aynı zamanda daha hızlı çözüme götürür.



Bu veri yapılarının farklı olmasının nedeni problemin çözümünü kolaylaştırmak. Karmaşık algoritmalar basit veri yapıları içerinde kurgulanabilir.

3)

Algoritma jenerektir. Birçok şeyi karşılar.

Adım adım çözüme götürmeyi hedefler.

Özel durum yoktur.

Tüm veriler için çıkışa gider.

Programlama dili için platform seçilir. O dile özgü syntax kullanılarak program yazılır.

4) İlkel bir toplama programı

```
def sumOf(n):
    sum=0
    for i in range(1, n+1):
        sum = sum+i
    return sum
```

1 işlem 1 zamandır. n kez yapılır.

$\frac{n.(n+1)}{2}$ ise 1 adım

işlemciye 1 kez müracaat eder.

Bu kod n kez çalışır, n kez işlemciye müracaat eder.

(n=5.000.000 olsun, yukarıdaki alp için) Yukarıdaki kod tübitak'ın bilgisayarında ve bizim Pentinium 3 32mb bilgisayarımızda tek adımdaki işleme koysak ortam aynı olmaz. Bu nedenle hangisi hızlı söylenemez.

Problem	Çözüm
$\Sigma 1 \ldots N$	1) + for kalıbı ile +

```
def sumOfN(n):
    sum=0
    for i in range (1, n+1):
        sum=sum+i
    return sum
```

```
def foo(tom):
    fred=0
    for bill in range (1, tom+1):
        barney=bill
        fred=fred+barney
    return fred
```

Kod kalitesi düşük / anlaşılması zor.

Soldaki gerekleme daha okunur, daha hızlı.

Bunları okunurluğa göre ölçecek olsak sağdaki daha az okunur.

Fakat algoritmaları okunurluğa göre kıyaslamıyoruz.

5)

CPU	RAM
İşlemciyi ne kadar yoran	Belleği ne kadar işgal eder

Biz bu zamana önem vereceğiz.

Out of Range (memory)

↳ Program çalışırken kitleniyor. CPU çalışabilecek durumda ama bellekte yer yok.

6) Benchmark Analizi:

Bilgisayar dünyasında benchmark performans değerlendirmeye verilen isimdir. Bu işi yapmak üzere özel olarak geliştirilen bilgisayar yazılımı, donanım bileşenini (işlemci gibi) veya başka bir yazılımı tekrar eden işlemlere maruz bırakarak geçen süre içerisinde performans ölçümleri yapar. Daha sonra kullanıcıya verilen detaylı analiz raporları, donanım veya yazılımın o süreçte nasıl tepki verdiğini gösterir. Bu işlem donanımın limitlerini görmek için yapılabileceği gibi üretim aşamasında dayanıklılık testlerinde de rol oynar.

GIPTA

Algoritmalarda benchmark testleri vardır.

1) Sıralı bir liste ile çalışınca sonuç nedir?

2) Ters sıralı bir liste ile çalışınca sonuç nedir?

3) Rastgele sıralanmış bir liste ile çalışınca sonuç nedir?

7) Yürütme Zamanı:

Bir algoritma çalışmasını bitirene kadar geçen süre yürütme zamanı olarak adlandırılır. Ve algoritmada genelde elemanı sayısı n olarak gösterilir ve yürütme zamanı da T(n) ile ifade edilir.

Algoritmadaki elemanı sayısı çok fazla olduğunda yürütme zamanı, zaman karmaşıklığı olarak adlandırılır. Ve derecesi asimptotik notasyon ile verilir. [O(n)] notasyonu

Algoritmanın işlevini yerine getirmesi için kullandığı bellek miktarına alan maliyeti denir. Örneğin n elemanlı bir dizi, elemanlarının her biri 4 byte ise bu dizi için alan maliyeti bağıntısı;

$S(n) = 4n$ 'dir.

Aynı şekilde alan karmaşıklığı da elemanı sayısı çok büyük olduğu zaman alan maliyetini de ifade eden asimptotik ifadedir. Zaman karmaşıklığında kullanılan notasyonlar burada da kullanılır.

8) Kıyaslama:

Kıyaslamada bellek ve yürütme zamanını dikkate alacağız.

9)

n değerinden bağımsız yürütme zamanı var. 1. çözüm yolunda n arttıkça yürütme zamanı da artar.

* Aynı algoritmayı → farklı platform / → farklı dil } uygulandığında farklı sonuçlar oluşur.

10) Big-O Notasyonu

Yürütme Zamanı = İşlem Sayısı X İşlem süresi
↓
(Adım)
(Temel İşlem)

* Kabul edilen her işlem 1 işlemci zamanını almaktadır *

$\boxed{T(n)}$
↓
(girdi olarak girilen depo) / (karmaşıklık derecesi)

dil + platform bağımsız

Algoritma Analiz
↓
Big-O
↑
Baskın
↓
Baskın Bileşen

(Sadece N'ye odaklı)

$T(n) \longrightarrow O(n)$

Yürütme Zamanı — Yürütme Zamanı Baskın Bileşen

11)

Bu kodda n+1 işlem yapılır n yeterince büyükse n'ye yakınsar kabaca bunun adım sayısı n'dir, diyebiliriz

12)

Problem	Çözüm
$T(N)$	$O(N)$
$N+1$	N
N^2+N	N^2
2^N+N^2	2^N

$O(f(n)) \longrightarrow$
- $O(1) \rightarrow$ Sabit
- $O(N) \rightarrow$ Doğrusal
- $O(N^2) \rightarrow$ Karesel
- ⋮

$O(f(n))$
↓
- $O(1)$
- $O(N)$
- $O(N^2) \rightarrow$ 2 döngü var. → İşlemci N^2 kez ziyaret ediliyor anlamında
- $O(2^N)$
- $O(\log N)$
- $O(N \log N)$
- ⋮

13) $T(n) \leq c_0 * f(n)$ $\quad n \geq n_0$ (c_0 ve n_0 sabit) varsa $O(f(n))$ vardır

$T(n) \leq c_0 . f(N)$ varsa $O(f(n))$ denilebilir

$N \geq N_0$

14)

$f(n)$	İsim	N	f(n) (sabit)	Log	doğ.	nlogn	n^2
1	sabit	1	1	0	1	0	1
$\log N$	logaritmik	10	1	1	10	1	100
N	doğrusal	100	1	2	100	200	$(100)^2$
$N \log N$	doğrusal logaritmik	1000	1	3	1000	3000	$(1000)^2$
N^2	karesel						
N^3	kübik						
2^N	üstel						

15)

Algoritma analizi için işlemciden bağımsız birşey sormalıyız

$T(n) = 3n^2 + 2n + 4$

$O(f(n)) = n^2$

G.S. 1 Bu problemleri nasıl çözeriz

```
def foo(N):
    if N<1:
        return 0
    return foo(N-1)+N
```

$O(n)=?$

özyineli — Bunu hesaplar

```
def foo(N):
    if N<1:
        return 0
    return foo(N/2)
```

$O(n)=?$

G.S.2 Algorithms, Sedgewick bak.

G.S. 3 Interactive Python → Algoritma Analiz

2 video - 6 G.S. - 5 programlama sorusu

Geçen sorulu G.S'lere bak.

Big-O Notasyonu

04/03/2013

Raziye AYDIN

$O(f(n))$

- 1 : sabit
- n : doğrusal
- n^2 : bublde
- logn : listelen aramasında kullanılan Binary Search
- nlogn : FFT, Quick Sort, Sıralama Algoritmaları

$c>1$ $\leftarrow c^n$

- n! : Brute Force

ÖRNEK: $6x^4-2x^3+5$ fonksiyonunu Big-O notasyonu ile gösteriniz

$6x^4-2x^3+5 \leq c \cdot O(f(n))$ $\quad n \geq n_0$

$|6x^4|-|2x^3|+5$

$6x^4+2|x^4|+5|x^4| = 13x^4$

487.258 → Big O notasyonu ile ilgili tartışma.

$n^2 + 2n + 1 \rightarrow \leq 4n^2 \rightarrow O(n^2)$

$2n \rightarrow 2n^2$, $1 \rightarrow 1n^2$

$(n+1)^3 \rightarrow (2n)^3 \rightarrow 8n^3 \rightarrow O(n^3)$

$n \rightarrow n$

$\sum_{i=1}^{n} i \rightarrow \sum^{n} n \rightarrow n^2$ (Her halükarda n^2'den küçük olduğu için Big-O'su n^2'dir.)

$\rightarrow O(n^2)$

$O(f(n))$ → üst sınır

Θ → teta → üst, alt

$T(P(n))$ her zaman $O(f(n))$'den küçüktür.

2 Anagram

Aynı dizginin harflerin sırası değişmiş halidir

heart → earth

ÖRNEK: İki dizginin anagram olup-olmadığını kontrol edeceğiz.

Bunun için 4 yöntem vardır.

1. Denetle

S1'deki tüm karakterleri S2'de denetle.

Boşlukları denetlemeleri None ile yerleştir.

hearth, earth olmuş olsaydı farklı karakterler tekrar edilmiş olurdu

↓ sıklık → histogram

Aralığımızda karışıklık olmaması için bulduğumuzu sildirebiliriz yoksa onun yerine bir işaret konabilir.

Denetleme - 23

```
def anagramSolution1(s1,s2):

    alist = list(s2)
    pos1=0
    stillOK = True

    while pos1 < len(s1) and stillOK:

        pos2 =0
        found= False

        while pos2 < len(alist) and not found:

            if s1[pos1] == alist[pos2]:

                found = True

            else:

                pos2 = pos2 +1

        if found:

            alist[pos2] = None

        else

            stillOK = False

        pos1 = pos1 +1

    return stillOK
```

Big-O notasyonu

```
1
1
1
N x [ 1
      1
      N x [ 1
            1
            1 ]
1
```

$4 + 4N + N^2 \rightarrow O(N^2)$

Algoritmanın en iyi durumu O(n) 'dir. Bunun olabilmesi için ilk stringin herhangi bir harfinin (tercihen ilk harf) diğerinde olmaması.

En kötü durum ise iki stringde sonuna kadar tarayacak olması. Bu durumda ikinci string birincinin tersi olmalı.

Ö.S.1 heart → haert
→ ahert
→ traeh

} için kaç adımda sonlanır

İpucu: found değişkeni sonlandığı anda döngünün kırıldığına dikkat edelim.

* Big-O notasyonu algoritmanın en kötü durumunu söyler. Kıyaslama yapılırken en kötü duruma bakılır. (O(n))

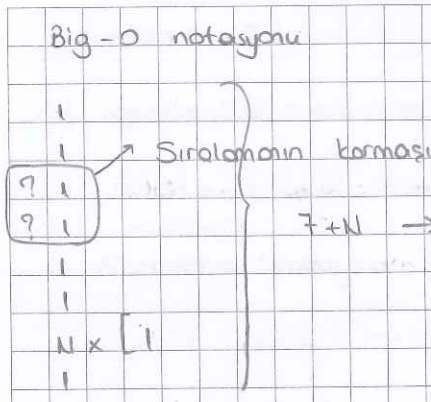
2. Sırala ve Karşılaştır

s1 ve s2 'deki karakterleri sırala. Sıralanmış halleri aynıysa anagramdır.

Gerçekleme - 26

```
def anagramSolution2(s1,s2):

    alist1 = list(s1)
    alist2 = list(s2)
    alist1.sort()
    alist2.sort()
    pos = 0
    matches = True

    while pos < len(s1) and matches:

        if alist1[pos] == alist2[pos]:

            pos = pos+1

        else:

            matches = False

    return matches
```

Big-O notasyonu



$7+N \rightarrow O(N)$

En kötü durum $O(N)$'dir

En iyi durum ise sıralanmış hallerinde ilk harfler farklı olunca $N=1$ olur.

Bu durumda $7+N = 8$ olur $\boxed{O(8)}$

$8(O(1))$

$\downarrow$ $\downarrow$

C Big-O $\rightarrow$ (sabit)

* Sıralama algoritmaları $O(N^2)$ ile $O(n\log n)$ ile değişir.
* Sıralamanın karmaşıklığı ne ise algoritmanın karmaşıklığı ona bağlıdır.

3. Kaba Kuvvet

Tüm olası değerleri sırala $\rightarrow$ s1'in tüm anagramlarını üret, üretilenler arasında s2 var mı, onu bul $\Rightarrow$ Brüt Force

ÖRNEK: heart $\rightarrow$ 5 karakter

5 4 3 2 1 $\rightarrow$ $5! = 120$ farklı anagramı vardır.

1. en kötü $\rightarrow$ s2'nin en sonuncusu olması.

2. en kötü $\rightarrow$ s2'nin s1'in anagramlarının içinde olmaması.

$\hookrightarrow$ Big-O'su $O(2^n)$ En kötü durum

* $n!$, n büyüdükçe 2^n'den daha hızlı büyür.

$T_n \leq c\, O(f(n))$

En iyi durum → s2'nin s1'in anagramlarının ilki olması → $O(1)$

* Algoritmaların sıralanması en kötü durumlara göredir.

4. Say ve Karşılaştır

s1 ve s2'deki karakter histogramını çıkart. İngiliz alfabesinde 26 karakter var.

a	1	✓
b	0	✓
c	0	✓
⋮	⋮	⋮
z	0	✓

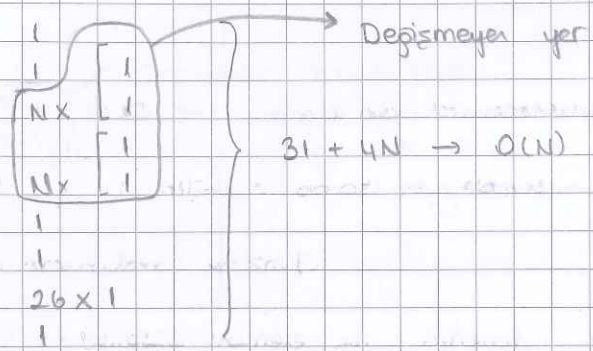
→ hepsi uyuyorsa anagramdır.

* ordinary fonksiyonu karakterin ASCII karşılığını verir.

Gerçekleme – 2.9

```
def anagramSolution4(s1,s2):

    c1 = [0]*26
    c2 = [0]*26

    for i in range(len(s1)):

        pos = ord(s1[i]) - ord('a')
        c1[pos] = c1[pos] + 1

    for i in range(len(s2)):

        pos = ord(s2[i]) - ord('a')
        c2[pos] = c2[pos] + 1

    j = 0
    stillOK = True

    while j < 26 and stillOK:

        if c1[j] == c2[j]:
            j = j + 1
        else:
            stillOK = False
    return stillOK
```

Big- O notasyonu



Karmaşıklık değişmeyecek ama zaman farkı var.

→ Döngüyü kurucak olan etken still OK'un false olması. O'da o'nın ya hiç olmaması ya da 2 kez olması durumunda geçerlidir.

③ Arama Algoritmaları

Ardışıl Arama (sequential)

Aranacak veri listede mi tek tek liste elemanlarına bakıyoruz.

Gerçekleme_33

```
def sequentialSearch(alist, item):

    pos = 0
    found = False


    while pos < len(alist) and not found:

        if alist[pos] == item:

            found = True

        else:

            pos = pos + 1

    return found
```

* Sabitler göz ardı edilir.

En kötü durum → Aradığımız eleman listede olmayabilir
→ Aradığımız eleman listede en sonda olabilir } O(N)

Yürütme zamanları da aynı.

En iyi durum → Aradığımız eleman listenin ilk elemanı olabilir → O(1)

Ortalama durum → $\frac{N}{2}$'den O(N)'dir.

Sıralı Liste

Aranacak Sayı = 50

Listemiz = 17 20 31 44 54 55 65 77 93

Ararken listenin sonuna kadar bakmaya gerek kalmaz. Yerinde bulunmadığı ilk yerden çıkılır (Aranan 50 değeri 44'ten sonra 54'ten önce olmalıydı. Devam etmeye gerek yok)

Bulma- Durma durumu

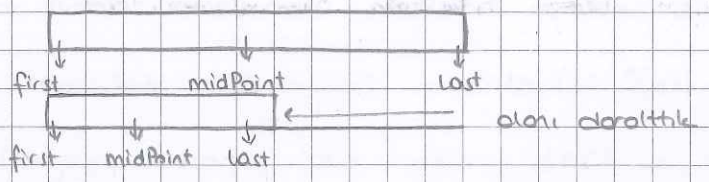
En kötü durum → En son elemana kadar gidip bulma/bulamama → O(N)

En iyi durum → ilk elemanda bulma → O(1)

* Sıralı listede erken çıkma hızı daha güçlüdür.

İkili Arama

Sıralı listenin üstünlüğünden yararlanabilmek için karşılaştırmanın daha zekice yapılması gerekir. İkili aramada ilk basta listenin ortasını belirle. Aradığın elemanı dizinin sol ya da sağ tarafından taraf seç tekrar ikiye böl. Sağ - sol belirle ve bulana kadar ya da liste tek elemana kadar küçülünceye kadar devam et.



→ Aradığımız değer soldaysa last sola kayar

→ Aradığımız değer sağdaysa first sağa kayar

Gerekleme - 41

```
def binarySearch(alist, item):

    first = 0
    last = len(alist) -1
    found = False

    while first <= last and not found:

        midpoint = (first + last) / 2

        if alist[midpoint] == item :

            found = True

        else :

            if item < alist[midpoint]:

                last = midpoint -1

            else :

                first = midpoint +1

    return found
```

N
N/2
N/4
N/8
⋮

$\sum N/2^i \longrightarrow \log N$

$\downarrow < N$

En kötü durum → Aradığımız sayı listenin en başında yada en sonunda ise → $O(\log N)$

En iyi durum → Aradığımız sayının listenin ortasında olması → $O(1)$

Yarılanma Sayısı : 3

$\downarrow$

$2^3 < 10$

$\downarrow$

listenin eleman sayısı

4 Hashing (Çırpı)

11/03/2013
Raziye AYDIN

Sözlük → Python

Hash → Ruby

İlişkisel Veri Tabanı → SQL

Çırpı → Hash

Türkiye	Ankara
Yunanistan	Atina

$O(1)$ "kişi sayısından bağımsız bir şekilde sonuca sabit bir adımda ulaşılması" anlamına geliyor.

Çırpı:

fonksiyon
↓ girdi

[Çırpı]

→ indis (tamsayı)

Github → status, comit, revision

$ git status

bu şekilde çalıştırılıyor.

* MD5 → CD'yi alıp çırpı fonksiyonundan geçirip sonuçta çıkan sayıyı göstermektedir.

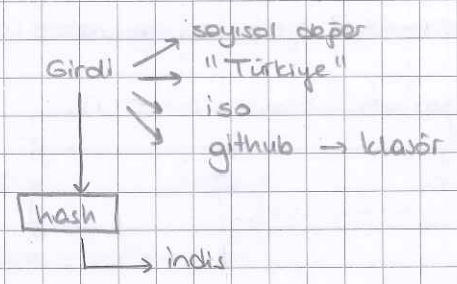
GIPTA

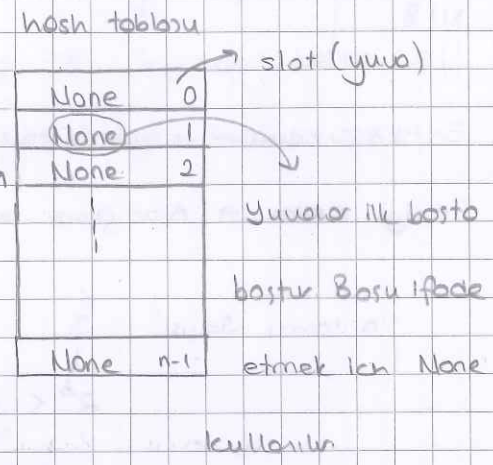
TODAY

* Kendimize ait MD5 kodu üretme

* Her eleman çarpanın dediği yere konumlanabiliyorsa (Türkiye dediğimizde Ankara diyorsa) karmaşıklığı 1'dir (O(1))

* hash tablosu satırlardan oluşur





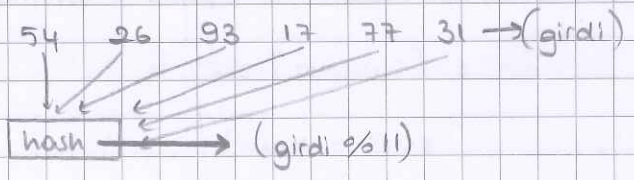
h= Hash() → Boş bir hash oluşturur

hash değerlerimiz [54, 26, 93, 17, 77, 31]

hash (2 geldiğini varsayalım)

hash'imizin max sayısı 11 olsun.

→ hash → (girdi %11) olsun dedik.



77	0
None	1
None	2
None	3
26	4
93	5
17	6
None	7
None	8
31	9
54	10

77 var mı diye arayacağız. Çarpıdan geçiriyoruz. 77'nin hash'ten indisi 0'dır. 0 numaralı yuvaya gidiyoruz. Var mı diye kontrol ediyoruz

hash'ten geçirmesi (mod'unu almak) → 1 adım

Tabloya yığması → sabit → 1 adım

* Doluluk oranının düşük olması, algoritmanın sıkıntılı olduğunu gösterir.

* Minimum bellek - maksimum doluluk

→ Aynı belleğe birden fazla değer geliyorsa ne yapacağız?

↓ hash'in en sıkıntılı yerlerinden biridir.

* Çakışmanın idare edilmesi çok önemlidir

* İyi bir hash 77'yi bir yere 55'i başka bir yere koyar. Buna mükemmel hash denir. Tablonun boyutu önemlidir. Tablo belleğin fazla kullanılmasına neden olur. Belleğin fazla kullanılması bellek israfına neden olur.

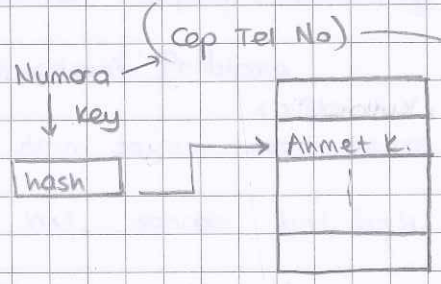
mükemmel hash → hiçbir çakışmanın olmadığı işlem

Folding Yöntemi

hash olmadığında en ilkel yapı olarak diziyi kullanırız

100 milyon → slot sayımız olsun.
100 → bizim kişi sayımız

$\frac{100}{100 \text{ milyon}}$ → 1 milyon slottan 1 tanesi dolu olacak.



Folding Yönteminde 1. Yöntem

→ 436-555-4601

Sayıyı 2'li gruplandırıyoruz

43-65-55-46-01

Topluyoruz → 210

%11'ini alıyoruz → 1 → 1. yuvaya kişiyi yerleştir

Folding Yöntemi 2. Yöntem

Sayıyı 2'li gruplandır

43-65-55-46-01

Ters çevir → 34-56-55-64-10

Topla → 219

%11'i → 10 → 10. yuvaya kişiyi yerleştir

* Başka bir yöntem:

Karesini al, ortadan 2 hane seç → mid-square

66, 44, 54, 26, 93, 17, 77, 31, 25, 62, 33

mod %11 'de 3 farklı eleman aynı yere işaret ediyor.

Ç.S. Diğer 3 yöntemle kaç tanesi aynı yere işaret eder / eder mi?

Bu yöntemlerden farklı bir hash tablosu yapıcaz

Ç.S. 11 değer, 11 slot var elimizde 11'de 11 yapan matematiksel (karmaşık olmayacak) ifadeyi çözünüz.

Karakter Tabanlı Çarpım İşlemi

Stringi oluşturan karakterler üzerinde çalışmanın ve karakterlerin ASCII değerlerini, değer olarak almanın

→ "cat" için:

ord('c') → 99
ord('a') → 97
ord('t') → 116

} (değerlerin toplamı) %(tablo boyutu)

→ "cat" ile "tac" ya da "act" nın ayrılması

ağırlıklandırma yaparak cat ile tac ya da act'yi kullanabiliriz

① ② ③
c a t ⇒ 1.c + 2.a + 3.t

* Mükemmel çarpma yakına çakışma olur

Çakışmanın Giderilmesi

Açık adresleme: Sayılar sırayla yerleştirilir. Bir elemanın dolu olan yere yerleşmesi gerekiyorsa, orası dolu olduğundan yerleşemeyecek. Bu nedenle o elemanı ilk boş yere yerleştiriz.

* Sayının gideceği yer doluysa aşağı doğru boş yer bulana kadar kayacak, yoksa en başa dönecek

Ö.S Aradığımız sayının hash kodu ile, sinadığımız hash kodu aynı mı diye kontrol eden programı yazınız

→ Atlatma Tekniği

Sonraki boşluğa yerleştirmek yerine n atla, çakışan elemanı araya yerleştir

* Bu yöntemin avantajı kümelemeyi önlemesi

Matematiksel olmaması pek güzel değildir Programcıların mantığına uygundur Biraz daha sezgiseldir

* Kümelemenin dezavantajı diğer gelenler için sürekli bir yer arama sıkıntısı olacak

Sonuçta ilk yerler doluysa sürekli bir kayma olacak (Sayıların hash kodlarının olduğu slotlara daha önce bir sayı (eleman) geldiyse)

n=1 atlama ⇒ açık atlama

n=3-5 atlama ⇒ atlama

→ Yeniden Girpme (Rehashing)

→ Quadratik Problama

Adım sayısını sabit tutmak yerine 1,3,5,7,--- artımlı yapmak

ilki h+1, sonraki h+1, h+4, --- şeklinde gidecek

→ Zincirleme Yöntemi

1. girp
2. slot doluysa zincirin sonuna ekle

} Daha makul bir sonuç üretebilir.

<u>ararken</u>

1. girp
2. slottaki zincire bak

[77, 66, --]
26
⋮
17
⋮

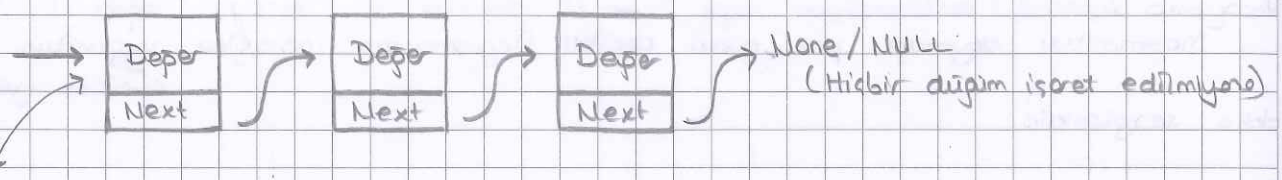
Bağlı Liste:

* Bütün değerler 1 slotta toplandıysa, bu en kötü durumdur. Hash kodu düzenlenmelidir.

→ Değişken uzunluklu liste kavramı

bağlı liste → Ağaç, Graf } Veri yapıları

Bağlı liste, üzerinde bir değer tutan ve kendinden sonrakini işaret eden en ilkel veriyapısının adıdır.



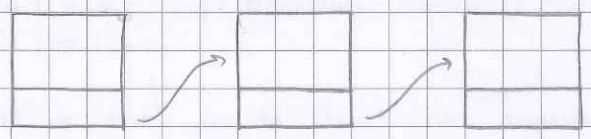
* Listenin içinde dolanabilmemiz için ilk değeri bilmemiz yeterli.

Null'a ulaşıncaya kadar devam edebiliriz.

Değer, Next değerinin pointer'ıdır. Next, düğümdür.

Yani bir düğüm yeni bir değere gider.

G.S.



Bir bağlı liste oluşturacağız. Bu listeyi hep sonuna ekleme olacak. (Sona ekleme yeteneğine sahip bağlı bir liste oluşturunuz)

Listenin Gücü

1. Ekleme: Bağlı listenin sonuna gidecek, yeni bir düğüm ekleyip pointer'ını NULL yapacak.

2. Arama: Düğümlerle dolaşacak. En sonunda NULL'a gelip yok diyecek.

0	1	2	3	4	5	6	7	8	9	10
	None	None					None	None		
77				26	93	17			31	54
44									20	
55										

Yukarıdaki ayrıştırma yöntemi gösterilmektedir. Bu yöntemde sıralama algoritmalarını uygulasak işimiz daha kolaylaşır. Ancak her hücrede en kötü durum n'dir. Sıralamada nlogn'dir.

18/03/2013

Raziye AYDIN

GEÇEN HAFTANIN AYNISI !!!!!

25/03/2013

Raziye AYDIN

5 Sıralama

Bazı algoritmalar sıralı koleksiyonlarda oldukça başarılıdır → Anagram
→ ikili arama

Çok sayıda sıralama algoritması vardır.

Sıralama algoritmaları bilgisayar bilimlerinde önemli bir yer tutar.

Eleman sayısı arttıkça sıralamada problem olur ancak karmaşık algoritmalar da az elemanlılarda başarısız olabilir.

1. Baloncuk (Bubble) Sıralama (Sort)

Eleman kümesindeki elemanların, ardışık olarak birbiri ile karşılaştırılması ile gerçekleşen sıralama algoritmasıdır. Karşılaştırmalar ikişerli olarak yapılır ve en büyük değer bir sonraki indise geçer.

Yani ilk-ikinci eleman kendi arasında kıyaslanır, gerekirse yer değiştirilir. Daha sonra ikinci-üçüncü kıyaslanır.

En iyi durum performansı → $O(n)$ 'dır. Küçük problemlerde işlem, her bir elemanın 1 defa yapıldığından en iyi durum performansı doğrusaldır.

En kötü durum performansı → $O(n^2)$'dır. $n < 1000$ olduğu sürece hız anlamında sorun yaşatmayacaktır.

54 26 93 17 77 31 44 55 20

26 54

26 54 93

26 54 17 93

26 54 17 77 93

26 54 17 77 31 93

26 54 17 77 31 44 93

26 54 17 77 31 44 55 93

26 54 17 77 31 44 55 20 93

ilk geçiş

ilk geçişte 1. en büyük sayı doğru yerindedir

26 17 54

ilk geçişte n-1 eleman sıralandı (93 kendi yerinde olduğunda)

Her bir adımda bir sonraki en büyük doğru yerine yerleşir

26 17 54 31 44 55 20 77 93

n-2 eleman sıralandı

listenin 2. en büyüğü

17 20 26 31 44 55 57 77 93 → küçükten büyüge sıralanmış hali.

Gerçekleme_7

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1, 0, -1):
        for i in range(passnum):
            if alist[i] > alist[i+1]:
                alist[i], alist[i+1] = alist[i+1], alist[i]
```

* Takas işlemi

python'da	diğer dillerde
a,b=b,a	temp = a a = b b = temp

→ Short Bubble (Sıralandığında bittir)

```
def shortBubbleSort(list):
    exchanges = True
    passnum = len(list) - 1
    while passnum > 0 and exchanges:
        exchanges = False
        for i in range(passnum):
            if list[i] > list[i+1]:
                exchanges = True
                list[i], list[i+1] = list[i+1], list[i]
        passnum = passnum - 1
```

2. Seçmeli (Selection) Sıralama

En basit sıralama algoritması olarak gösterilir. Elimizdeki dizide sıralanması gereken n tane sayı olsun. Bu sayıları küçükten büyüğe sıralamak gerekirse, sıralama algoritması şöyledir:

→ Dizinin 1. elemanından başlayarak, tüm elemanlarını kontrol ederek en küçük sayıyı bul

→ Bulduğun en küçük sayıyı, dizinin 1. elemanı ile yerdeğiştir (swap)

→ Dizinin 2. elemanından başlayarak, tüm elemanlarını kontrol ederek en küçük sayıyı bul

→ Bulduğun en küçük sayıyı, dizinin 2. elemanı ile yerdeğiştir (swap)

⋮

* Her bir yineleyiştaki en küçük sayıyı bulup onu swap ettiğimiz yer, o sayının sıralanmış dizideki uygun yeri olmuş olur.

→ Karmaşıklık ←

Sıralanmış dizinin her bir elemanını bulurken, sıralanmamış dizyi baştan sona taramamız gerekiyor. 1. en küçük elemanı bulurken → n-1 tane kontrol yapmamız gerekiyor

2. en küçük elemanı bulurken → n-2 tane kontrol yapmamız gerekiyor

⋮

n-1. en küçük elemanı bulurken → 1 tane kontrol yapmamız gerekiyor

Yapılan tüm kontrollerin sayısı $\frac{(n-1)\cdot n}{2}$ olduğundan karmaşıklık $O(n^2)$'dir

54 26 93 17 77 31 44 55 20 (n-1) kıyaslama

17 26 93 54 77 31 44 55 20 (n-2) kıyaslama

17 20 93 54 77 31 44 55 26

⋮

17 20 26 31 44 54 55 77 93 1 kıyaslama

→ Bubble Sort ile aynı sayıda kıyaslama yapılır. Karmaşıklık $O(n^2)$ 'dir.

→ Yerdeğiştirme miktarı daha azdır.

→ Yürütme zamanı daha kısadır.

3. Araya Girmeli (insertion) Sıralama

Araya girme algoritması, sıralanacak dizinin ilk elemanını yerinde bırakarak ondan sonraki elemanları, sırayla alarak, sıraya uygun yere koyar.

54 26 93 17 77 31 44 55 20

26 54 93 17 77 31 44 55 20 → 26 yerleşti → 1. geçiş

26 54 93 17 77 31 44 55 20 → 93 yerleşti → 2. geçiş.

17 26 54 93 77 31 44 55 20 → 17 yerleşti. → 3. geçiş

17 26 54 77 93 31 44 55 20 → 77 yerleşti → 4. geçiş

⋮

17 20 26 31 44 54 55 77 93 → 20 yerleşti → 8. geçiş

→ Altlisteler birer birer büyür ancak sıralı şekilde büyür.

* 5. geçişin ayrıntıları *

* "31" sayısını nereye koyacağız

* Sıralı altlistede en büyük değerden başlayarak solda daha kendinden küçük olan ilk değerin sağına yerleştirilir.

17 26 54 77 93 31

31'den küçük ilk sayı. (Sağına yerleştireceğiz).

17 26 31 54 77 93

altlişte sıralandı.

Gerekleme - 19

```
def insertionSort (alist):

    for index in range (1, len(alist)):

        currentvalue = alist[index]
        position = index

        while position > 0 and alist[position-1] > currentvalue:

            alist[position] = alist[position-1]
            position = position-1
        alist[position] = currentvalue
```

→ Karmaşıklık ←

En kötü durum → $O(n^2)$ 'dir

En iyi durum → $O(n)$ 'dir

* Takas işlemi / 3 = Kaydırma işlemi ⇒ Süre bazında düşünürsek

4. Kabuk (Shell) Sıralama

Araya girmeli sıralamanın iyileştirilmiş halidir

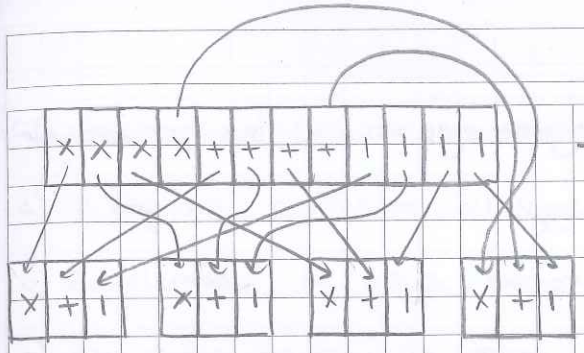
Mantık olarak

→ Listeyi alt listelere böl
→ Herbir alt listeyi insertion sort ile sırala.
→ Sıralanan alt listeleri birleştir ve tekrar insertion sort ile sırala.

? Alt listelere nasıl böleceğiz? → Tecrübesel, sezgisel ←

Ardışıl (tek tek) yerine atlamalı seçmeliyiz

Böl → Sırala → Birleştir → Sırala



→ Asıl liste

→ Altlıdı olarak altlistelere ayırdık

→ Her altliste kendi içinde sıralanıp birleştirilir ve tekrar sıralanır.

54 | 26 | 93 | 17 | 77 | 31 | 44 | 55 | 20

Alt listelerimiz

1 → 54 17 44
2 → 26 77 55
3 → 93 31 20

} Alt listeleri kendi aralarında sıralı.

1 → 17 44 54
2 → 26 55 77
3 → 20 31 93

↓

17 26 20 44 55 31 54 77 93 sıraylı yerleştirilir

altlisteler birleştirildikten sonra insertion (1 atlama) sıralama ile sıralanır.

17 20 26 44 55 31 54 77 93

17 20 26 31 44 55 54 77 93

17 20 26 31 44 54 55 77 93 → Sıralı listemiz

Toplamda

3 tane → altliste sıralama

1 tane → altlistelerin sıralanıp birleştirildiği büyük listeyi sıralama

+

4 tane insertion sort yaptık.

→ Boyut Yarılama

Altlisteler, özyinelemeli bir şekilde boyut yarılamayla oluşturulup, sıralanabilir

Bölme sayımız → $\frac{N}{2}$ → listedeki eleman sayısı

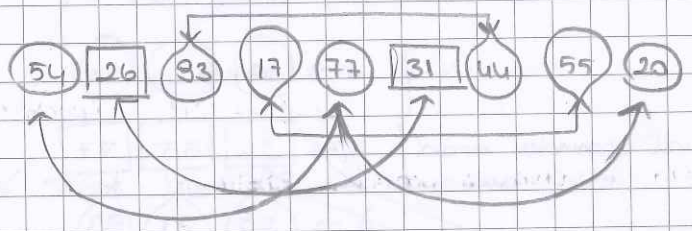
	α	1	2	3 *	α 4			*
54	26	93	17	77	31	44	55	20
*	1	2	3	4	1	2	3	4

$N = 9$ $\quad \frac{N}{2} = 4$ (Tamsayı)

↳ Bu sayı'da bir tekrar

1 altliste belirlendi

Altlistelerin son halleri:



Gerçekleme_29

```
def shellSort(alist):
    sublist = len(alist) /2
    while sublistcount > 0:
        for startposition in range(sublistcount):
            gapInsertionSort(alist, startposition, sublistcount)
        print "After increments of size", sublistcount, "The list is", alist
        sublistcount = sublistcount /2

def gapInsertionSort(alist, start, gap):
    for i in range(start+gap, len(alist), gap):
        currentvalue = alist[i]
        position = i
        while position >= gap and alist[position-gap] > currentvalue:
            alist[position] = alist[position-gap]
            position = position - gap
        alist[position] = currentvalue
```

⇐ Tekrar Dön ⇒

* Karmaşıklık → Listeyi yarılayarak ilerlersek $O(n^2)$'dir

Ç.S. Herhangi bir liste içerisinde dolaşacağız

a) 2 atlayarak oluşturulan alt listeyi ekrana yazdır

b) Bu listedeki en büyük elemanı ekrana yazdır

c) Alt listeyi insertion ile sırala

Ç.S. gap-insertion sort programını elimizdeki listede

boşlangıç değeri = 0 } için takip edecağiz
gap değeri = 2

* insertion sort ile shell sort hemen hemen aynıdır.

Sınavda Sorulabilir !!!

Verilen liste 3 atlamalı 3 altlisteye bölünerek kabuk sıralamaya tabi tutulacaktır

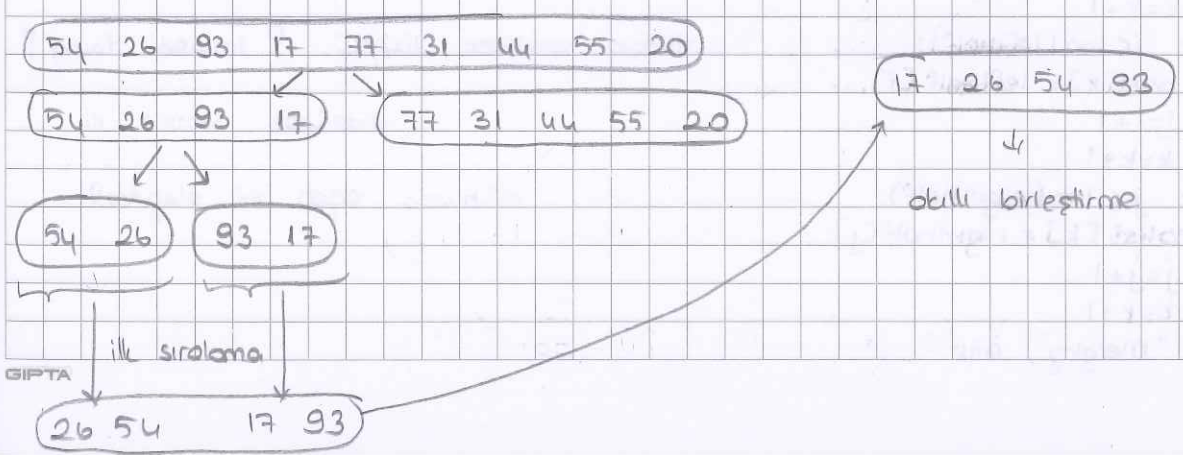
a) Bu liste üzerinde çalışırken en fazla karmaşıklığı üretecek dizilim?

b) Karmaşıklığı en aza indirgeyen dizilim?

5. Birleştirmeli (Merge) Sıralama

Merge Sort algoritması divide and conquer (böl ve yönet) metodunu kullanarak verileri sıralar.

Algoritma verileri, tek eleman kalıncaya kadar recursive fonksiyonlar yardımıyla sürekli iki'ye bölen. Tek kalan elemanlar ise kendi aralarında sıralı bir şekilde birleştirmeye başlar. Tek kalan elemanlar birleştiğinde de dizi sıralanmış olur.



Adım adım 1/2'lik düzeyde bölüp akıllı sıralama ile birleştiriyoruz

Tek eleman kalıncaya kadar bölmeni + bir araya getirirken sıralı şekilde birleştirmen

→ işlemler 2 adımda tamamlanır

C.S. Parçalama işlemini yaptıracak kodu yazınız

C.S. Merge Sort'un parçalama işlemlerde her bir kattaki elemanı ya da dizayı yazdır

C.S. Tek tek alınan listeleri birleştirme işlemi yaptıracak kodu yazın

Gerçekleme_36 (Split)

```
def mergeSort(alist):
    print "Splitting", alist
    if len(alist)>1:
        mid = len(alist)/2
        lefthalf = alist[:mid]      } Listenin sol-sağ parçalarını
        righthalf = alist[mid:]     } belirledik

        mergeSort(lefthalf)         } Sağ ve sol parçaları kendi
        mergeSort(righthalf)        } aralarında sıraladık
```

Gerçekleme_37 (Merge)

```
i,j,k = 0,0,0
while i< len(lefthalf) and j< len(righthalf):
    if lefthalf[i] < righthalf[j]:
        alist[k] = lefthalf[i]
        i=i+1
    else:
        alist[k] = righthalf[i]
        j=j+1
    k=k+1
while i< len(lefthalf):
    alist[k] = lefthalf[i]
    i=i+1
    k=k+1
while j< len(righthalf):
    alist[k] = righthalf[j]
    j=j+1
    k=k+1
print "Merging", alist
```

→ Karmaşıklık ←

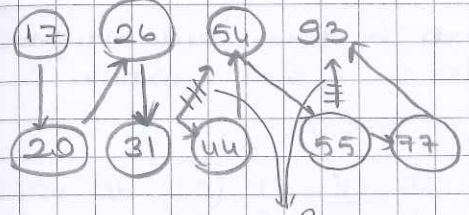
İkili aramada → O(logn)

Yarılama (split) → O(logn)

Birleştirme (merge) → O(n)

Shell karmaşıklığı → O(nlogn)

GS. Elimizde 2 tane sıralı liste olsun. Bu iki sıralı listeyi en az karşılaştırmayla akıllı birleştirecek program



Bu yollar geçerli olur. 31'den sonra 44 gelir. 44'ten sonra 54'e gidilir.

GS. Merge Sort'u extra bellek ihtiyacı olmayacak şekilde tasarlayınız

01/04/2013 Raziye AYDIN

6. Hızlı (quick) Sıralama

Hızlı sıralama da böl ve yönet mantığıyla çalışmaktadır. Yarılama yerine pivot seçimi yapılmaktadır.

→ pivot değeri seç.
→ liste içerisinde pivotun doğru konumunu bul (bölme noktası)
→ bölme noktası yardımıyla listeyi ikiye böl (sol-sağ)
→ bu iki parçayı ayrı ayrı özyineli quick sort ile sırala.

? pivot seçimi ? Değişik şekillerde olabilir.

İlk eleman seçilebilir.

Rastgele bir değer seçilebilir

Quick Sort Algoritmasının adımları:

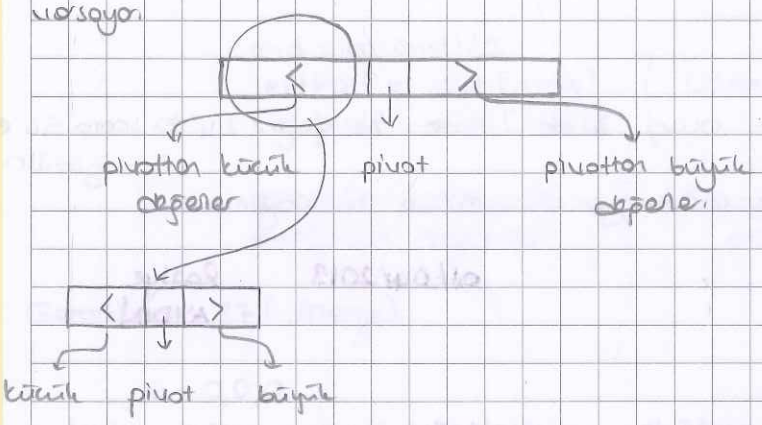
→ Sayı dizisinden herhangi bir sayıyı pivot eleman olarak seç

→ Sayı dizisini pivottan küçük olan tüm sayılar pivotun soluna, pivottan büyük olan tüm sayılar pivotun sağına gelecek şekilde düzenle (pivota eşit olanlar her iki yana da gelebilir)

Bu bölümlendirme işleminden sonra eleman sıralanmış son dizide olması gerektiği yere gelir. Algoritmanın bu aşamasına <u>bölümlendirme aşaması</u> denir.

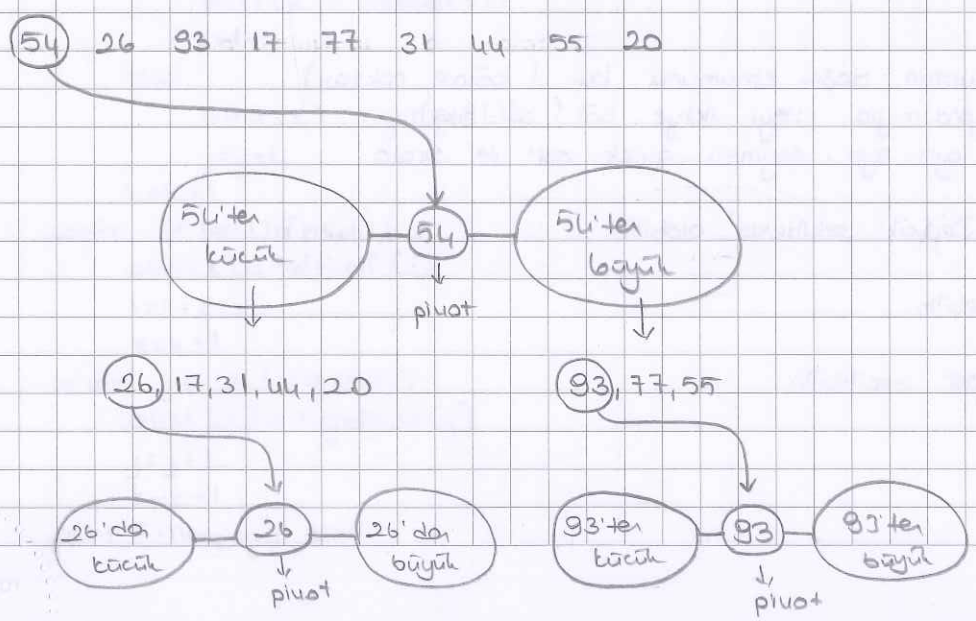
→ Pivotun sol ve sağ yanında olmak üzere oluşan iki ayrı küçük sayı dizisi, hızlı sıralama algoritması bu küçük parçalar üzerinde yeniden özyineli olarak çağrılarak sıralanır.

Algoritma içinde sayı kalmayan bir alt diziye ulaştığında bu dizinin sıralı olduğunu varsayar.



Elimizdeki listemiz: → (ilk elemanları pivot seçiyoruz)

54 26 93 17 77 31 44 55 20



pivot

54 26 93 17 77 31 44 55 20

i=1 → ← j=8

i 54'ten büyük bir değer bulununcaya kadar ilerliyor.

j 54'ten küçük bir değer bulununcaya kadar ilerliyor. (ilk adımda karşıdır ki 54'ten küçük olduğundan olduğu yerde kalır)

i 1. adımda 1 basamak sağa kaydı.

1 adımda → i=2 oldu
j=8 (olduğu yerde kalır)

Bunlar 1. adım. Önemli olan 2. adımdır ★★

2. adımda i ve j yer değiştirir (swap edilir)

swap edildi

54 26 20 17 77 31 44 55 93

i=2 → ← j=8

i 54'ten büyük bulununcaya kadar } i=4
j 54'ten küçük bulununcaya kadar } j=6
} swap edilir

swap edildi

54 26 20 17 44 31 77 55 93

i=4 j=6

54 ile 31'in durumuna bakacağız. Ortada kalan sayı pivottan küçükse pivotla swap edilir.

31 26 20 17 44 — 54 — 77 55 93 ⇒ ayrılmış olduk

Gerekleme_50 (Sort)

```
def quickSort(alist):
    quickSortHelper(alist, 0, len(alist)-1)

def quickSortHelper(alist, first, last):
    if first < last:
        splitpoint = partition(alist, first, last)

        quickSortHelper(alist, first, splitpoint-1)
        quickSortHelper(alist, splitpoint+1, last)
```

Gerekleme_51 (Partition)

```
def partition(alist, first, last):
    pivotvalue = alist[first]

    leftmark, rightmark = first+1, last

    done = False

    while not done:

        while leftmark <= rightmark and alist[leftmark] < pivotvalue:

            leftmark = leftmark + 1

        while alist[rightmark] > pivotvalue and rightmark >= leftmark:

            rightmark = rightmark - 1

        if rightmark < leftmark:

            done = True

        else:

            alist[leftmark], alist[rightmark] = alist[rightmark], alist[leftmark]

    alist[first], alist[rightmark] = alist[rightmark], alist[first]

    return rightmark
```

Ç.S. Elimizdeki listenin ters sıralı olması durumunda hızlı sıralamanın sonucu ne olur? (algoritmayı takip edin)

Ç.S. Sıralı listede pivotu ne seçersek tam yarılanır (en iyi performans)?

* En kötü durum → 93 20 26 31 44 54

⇒ Bütün elemanlar taranır, bu yüzden en kötü durum n log n'dir.

⇒ Tüm elemanları dolaşmanın maliyeti N'dir.

* Ortadaki pivot olacak şekilde

1 baş - 1 orta - 1 son } elemanlar seçilir. Bu elemanlar kendi aralarında sıralanır.

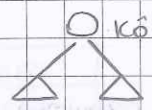
* Liste sıralıyı yakınsa çok iyi sonuç verir.

ödev quicksort gerçekleşmesinde pivot seçimini değiştirin

Ç.S. Parametre olarak verilen listenin medyan değerini hesaplayan algoritmayı yazınız

08/04/2013 Raziye AYDIN

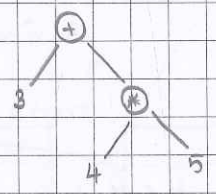
Ağaç Veri Yapısı



T = [⊕, [sol], [sağ]]

[⊕, [sol], [sağ]] [⊕, [sol], [sağ]]

Örnek:



[+ , [3], [* , [4], [5]]]

0 1 2

0 1 2

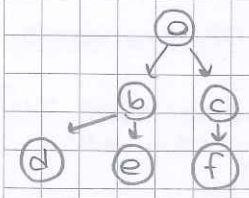
yapımızın köküne erişmek istersek → T[0]

ağacın köküne erişmek istersek → T[1][0] → Kökü temsil eder.

5'e ulaşmak istersek → T[2][2][0]

ilk kökün 2. elemanı

ağacın 2. elemanı



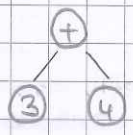
[a, [b, [d, [], []], [e, [], []]]

[c, [f, [], []], []]

]

! Kural belirlemeliyiz. Bizim kurulumuz ilk eleman soldadır.

Programlama Sorusu: Elimizdeki yukarıdaki gibi bir liste olsun köklerde dolaşarak bu kökleri ekrana yazdıran program (özyinelemeli fonksiyon ile çözülebilir.)

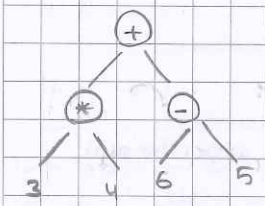


+ 3 4 → prefix → kök, sol, sağ

3 + 4 → infix → sol, kök, sağ

3 4 + → postfix → sol, sağ, kök

Sınav Sorusu:



prefix ile

+ * 34 - 65

Bunun program halli sınavda gelebilir. !

→ Altağaç Ekleme ←

Sol altağaç ekleme

```
def insertLeft(root, newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [newBranch, t, []])
    else:
        root.insert(1, [newBranch, [], []]
    return root
```

Sol eleman = 1

Bu ifadenin çıkarılması işlemi

Bu ifade çıktığında

root = [+, []] olur

root = BinaryTree(+) → [+, [], []]

branch = BinaryTree(3) → [3, [], []]

⇒ [+, [], []]

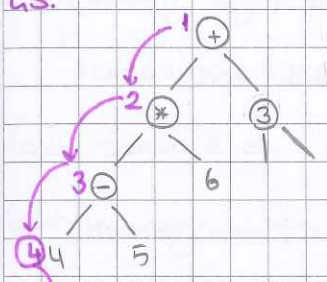
0 1 2

branch = [3, [], []]

Sağ altağaç ekleme

```
def insertRight(root, newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2, [newBranch, [], t])
    else:
        root.insert(2, [newBranch, [], []])
    return root
```

Ç.S.



1) Bu şekilin liste gösterimini yazınız

2) 3 fonksiyon tanımladık (BinaryTree, insertLeft, insertRight)

Bu 3 fonksiyonu kullanarak yandaki ağacı oluşturun.

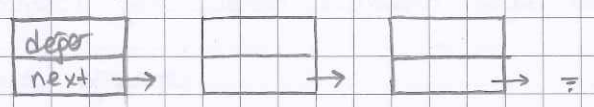
3) Elimizde böyle bir ağaç varsa

→ En solundaki değeri (4) söyleyecek programı yazınız

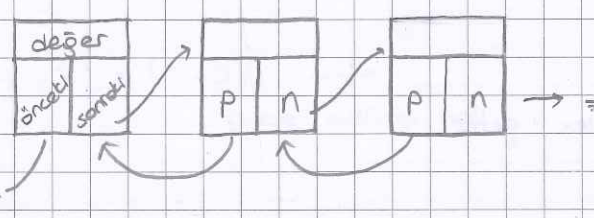
→ Sol dibe gidinceye kadar kaç yer gezilir (Bu ağacın yüksekliği nedir)? (4)

4) Solun sonuna 99 değerini eklemek istiyoruz. Ekleyen programı yazın.

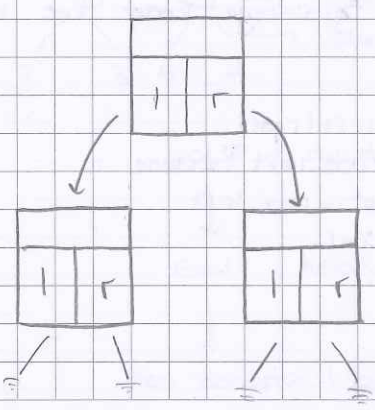
Düğümler ve Referanslar

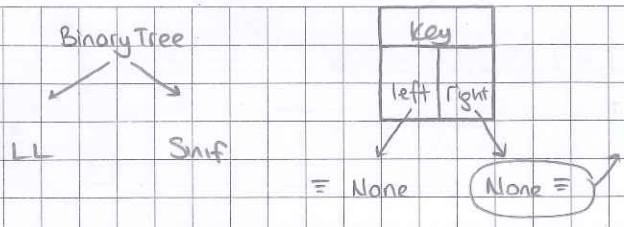


Burda sürekli ileri gideriz. Geriye gidemeyiz.



Double List. Biraz daha düzenlenek





G.S. Bundan önceki insert ile çağırıyorduk. Bundan sonra eklemleri append ile yapacak olsak nasıl kodlardık?

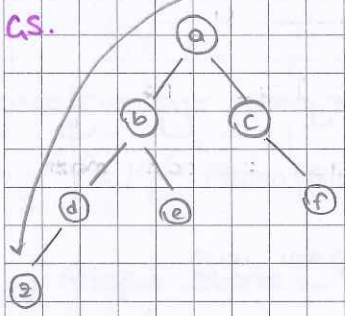
G.S. Elimizde 1 ağaç olsun. Bu ağacın en sol ucuna kadar olan kısmı döndürecek kodu tasarla.

(Mantık: Solun solunda birşey varsa sola geç, yoksa kaldığın yeri yazdır.)

G.S. En dibe ulaşıncaya kadar kaç düğümden geçer, bunu yazdıran kod.

G.S. En dip düğümün altına 2'yi ekle (Soldakine, sola ekle)

G.S. Yandaki ağaçta toplam kaç düğüm var, bunu yazdıran kodu yazın.



G.S. Düğümlerdeki değerlerin toplamını yazdıran program

Örnek kodlar:

a) En dibe giden:

```
def deep_left(root):
    while root.left != None:
        root = root.left
    return root
```

b) En dibe gidinceye kadar kaç kökten geçer:

```
i=1
def deep_left(root):
    while root.left != None:
        root = root.left
        i=i+1
    return i
```

c) En son sol dip düğümünün altına '3' yü ekleme

```
def deep_left(root):
    orroot = root
    while root.left != None :
        root = root.left
    root.insert('3')
    return orrood
```

15/04/2013 Raziye AYDIN

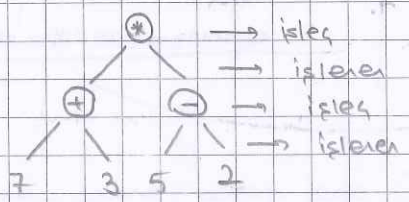
Parse Ağacı

Parçalama işlemi (parsing) bilgisayar bilimlerinde çeşitli amaçlar için kullanılmak tadır özellikle de dil ile ilgili işlemlerin hemen hepsinde ihtiyaç duyulan bir işlemdir. Örneğin bir programlama dilinde yazılan komutların algılanması için öncelikle kelimelerin parçalanması (parse) gerekir. Benzer şekilde doğal dil işleme (natural language processing) işlemlerinde de doğal dilde bulunan kelimelerin algılanması bir parçalamadan (ek ve köklerin ayrılmasından) sonra gerçekleşmektedir.

Çeşitli sebeplerle kullanılan parçalama ağaçları basitçe verilen bir dilbilgisine (grammer) göre verilen cümlenin (veya kelimenin) nasıl parçalandığını şekilsel olarak gösteren ağaçlardır.

Biz matematiksel (aritmetik) parse ağacını kullanacağız

$((7+3) * (5-2))$ ⟶ parse edilmiş hali



* Her işleç önce işlenenleri hesaplar.

parse (ayrıştır)
↓
eval (hesaplo)
↓
ters dönüştür. (Sadeleşmiş ifadenin infix notasyonu ile gösterimi.

$((7+3)*(5-2))$ → Her bir ifade <u>token</u> 'dur.

token listemiz : soğ ve sol parantezleri, işleçleri ve işlenenleri içerir.

sol parantez : yeni bir ifade başlatır, yeni ağaç oluşturur.

soğ parantez : ifade sonlandırır.

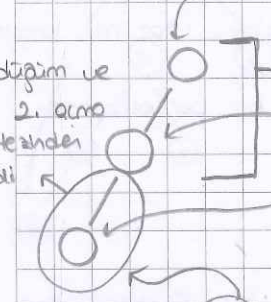
işlenenler : yaprak düğümlerindedir ve işleçlerin çocuklarıdır.

* her işleçin soğ ve sol çocuğu vardır.

Aşamalar

1. token : '(' → <u>Şimdiki düğümün</u> <u>sol çocuğu olarak yeni düğüm ekle</u>

→ <u>Sol düğüme geç</u>



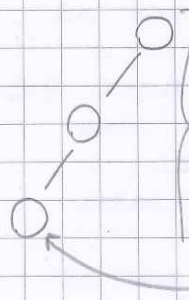
Gerçekleme_31

```
currentTree.insertLeft('')
pStack.push(currentTree)
currentTree = currentTree.getLeftChild()
```

2. token : '(' → 1.'nin aynısı

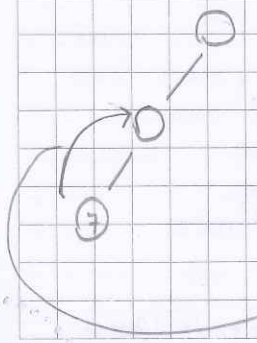
3. token : 7 (işlenen) → Şimdiki düğümün köküne işleneni koy

→ Ebeveyne çık.



Bu boşta oluşturduğumuz ağaç

<u>7 rakamını şimdiki düğümün köküne koymak</u>



7'yi yerleştirdik <u>Ebeveyne çıkıyoruz</u>

Gerçekleme_32

```
currentTree.setRootVal(eval(i))
parent = pStack.pop()
currentTree = parent
```

ilk aşamada işlenen gelene kadar yığıta atıp işlenen gelince yığıttan çek

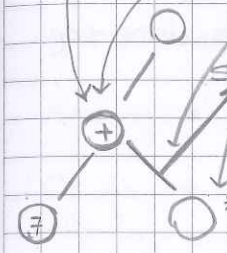
(Düğümlerin kökleri)

GIPTA

4. token : + (işleç) → şimdiki düğümün köküne işleci koy

→ sağa yeni düğüm ekle

→ sağ düğüme geç.



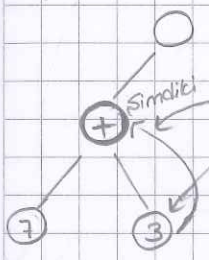
7'yi yerleştirip, 7'nin ebeveynine çıkmıştık (Bir önceki adımda)
Sağa yeni bir düğüm ekledik

Gerçekleme_33

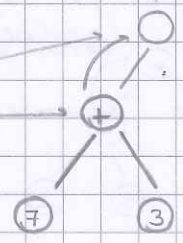
```
currentTree.setRootVal(i)
currentTree.insertRight('')
pStack.push(currentTree)
currentTree= currentTree.getRightChild()
```

5. token : 3 (işlenen) → 3.'nün aynısı → şimdiki düğüme işlenen koy

→ Ebeveyne dön.



6. token : ')' → şimdiki düğümün ebeveynine dön



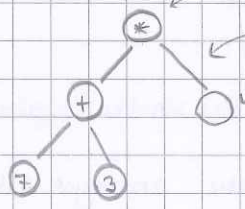
Gerçekleme_34

```
currentTree = pStack.pop()
```

7. token : * (işleç) → 4.'nün aynısı → şimdiki düğümün köküne işleci koy

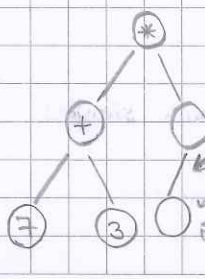
→ sağa yeni düğüm ekle

→ sağ düğüme geç.



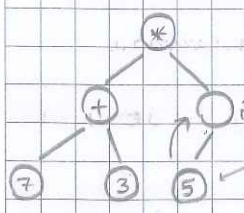
8. token : '(' → 1. aynısı → şimdiki düğümün solunu düğüm ekle

→ sol düğüme geç.



9. token : 5 (işlenen) → 3.'nün aynısı → Şimdiki düğümün köküne işlenen koy

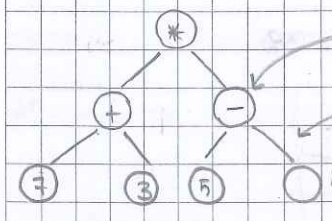
→ Ebeveyne çık



10. token : - (işleç) → 4.'nün aynısı → şimdiki düğümün köküne işleci koy

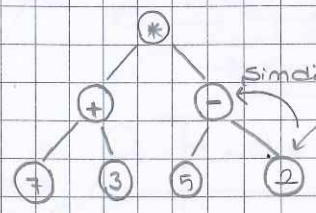
→ Sağa yeni düğüm ekle

→ Sağ düğüme geç

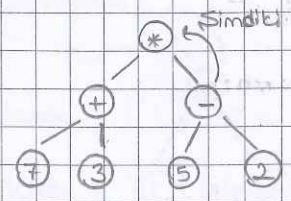


11. token : 2 (işlenen) → 3.'nün aynısı → Şimdiki düğümün köküne işlenen koy

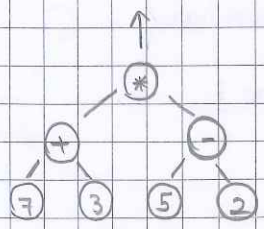
→ Ebeveyne çık



12. token : ')' → 6.'nın aynısı → Şimdiki düğümün ebeveynine dön.



13. token : ')' → 6.'nın aynısı → Şimdiki düğümün ebeveynine dön



~ Genelleme ~

* Sol parantez : Solo bir düğüm ekler. Bu şimdiki düğüm olur.
* İşlenen : şimdiki düğüme işleneni bırakıp ebeveyne çıkar.
* İşleç : İşleci şimdiki düğüme koyup, sağ düğüm oluşturup şimdiki düğümü sağ düğüm yapar.
* Sağ parantez : Şimdiki düğümün ebeveynini şimdiki düğüm yapar.

GIPTA

ÇS. Herhangi bir aritmetik ifadenin soldan sağa doğru taranması durumunda ağacın gelişimini ve currentTree (duruma göre "şimdiki kök") 'leri işaretleyecek (çizecek) çizimi yapınız.

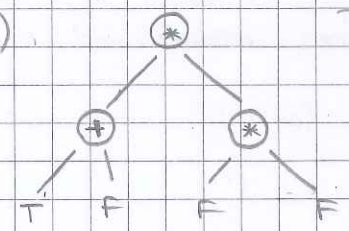
Mantıksal ifadeler

And → *

Or → +

} True / False

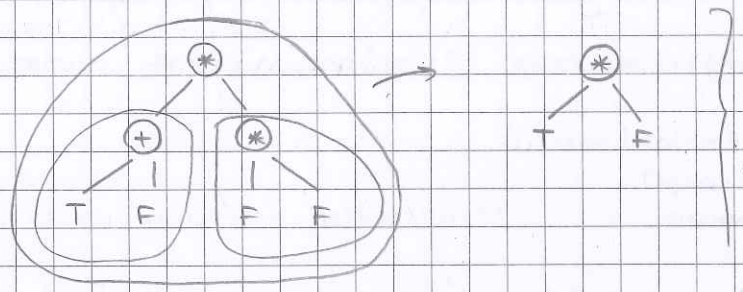
((T+F) * (F*F))



} mantıksal ifadenin ağacı.

ÇS. a) Kullanıcıdan almış olduğu mantıksal ifadeyi ayrıştırıp ağacını üreten kod

b)



} İndirgeme, ağacın küçültülmesi

* Biz bunu nasıl yaparız (kendi yöntemimizle, mantıken tasarlama)

c) a şıkkının aynısı, kullanıcıya "and" için birşey yazmamasını söyleyecek

(T+F)FF

and için "*" işareti kullanılmayacak

"+" işareti or'u gösterecek

Gerekleme_37

```
def buildParseTree(fpexp):

    fplist = fpext.split()
    pStack = Stack()
    eTree = BinaryTree('')
    p.Stack.push(eTree)
    currentTree = eTree

    for i in fplist :

        if i == '(' :

            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()

        elif i not in '+-*/)' :

            currentTree.setRootVal(eval(i))
            parent = pStack.pop()
            currentTree = parent

        elif i in '+-*/' :

            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()

        elif i == ')' :

            currentTree = pStack.pop()

        else :

            print " error : I don't recognize" + i

    return eTree
```

---

pStack() → ebeveyni tutmak için kullanılan yığıt

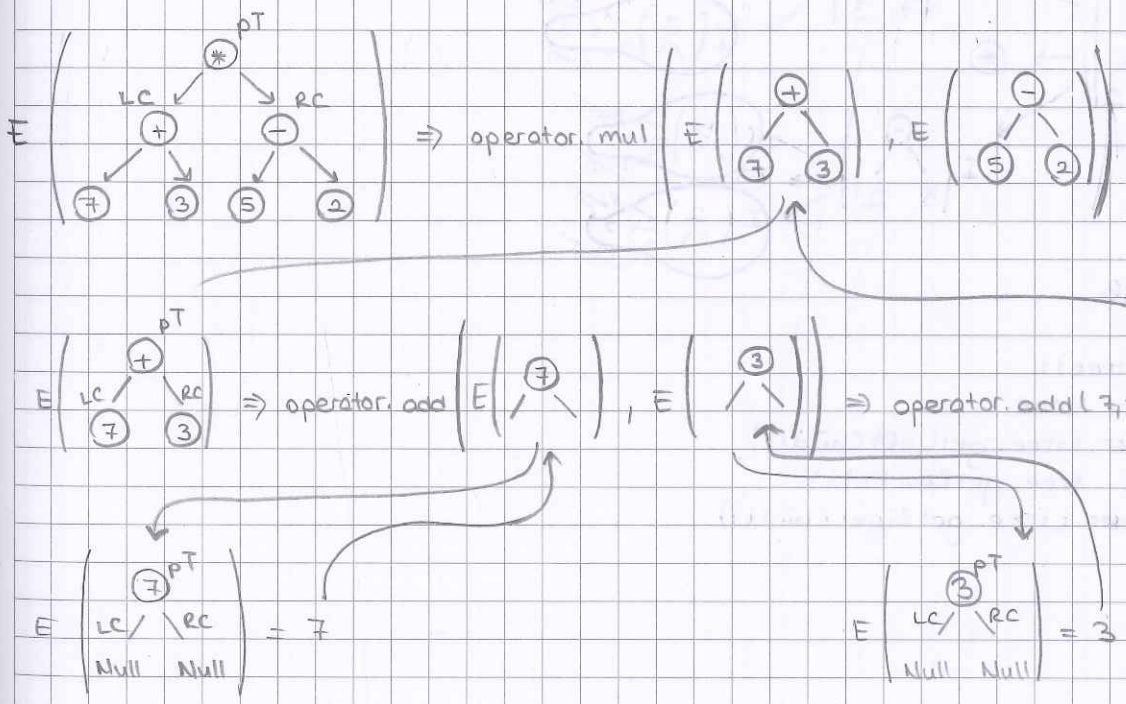
alt düğüme atlarken, şimdiki düğüm yığıta itiliyor [pStack.push(currentTree)]

ebeveyne dönmek gerektiğinde yığıttan çekiliyor [parent = p.Stack.pop()]

Parse Tree Hesaplayıcı

Gerekleme_39

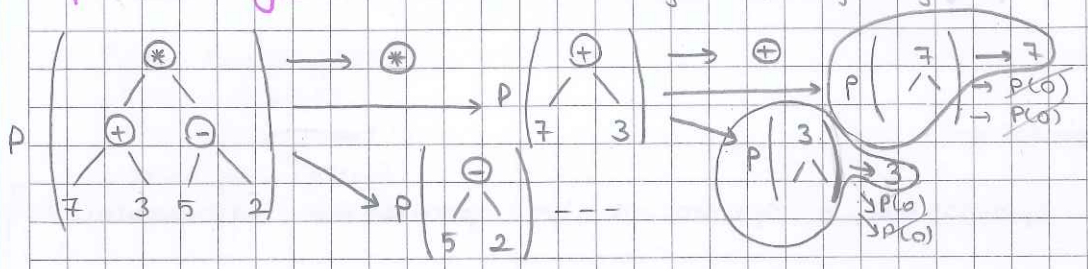
```
def evaluate(parseTree):
    opers = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.div}
    leftC = parseTree.getLeftChild()
    rightC = parseTree.getRightChild()
    if leftC and rightC:
        fn = opers[parseTree.getRootVal()]
        return fn(evaluate(leftC), evaluate(rightC))
    else:
        return parseTree.getRootVal()
```



Ç.S. Yukarıdaki gibi, örnek bir ifade üzerinde evaluate adımlarını şekil çizerek anlatınız, oklarla gösteriniz.

ağaçta dolaşım:

preorder gösterim: kök → sol altağaç ⇒ sağ altağaç



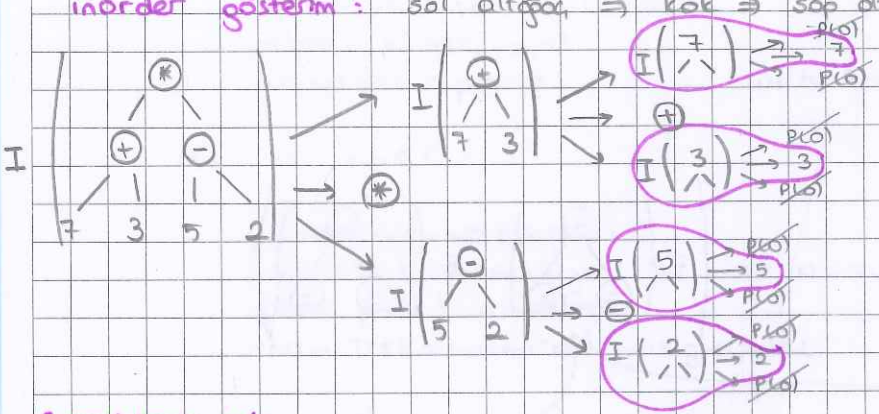
Gerçekleme_42

```
def preorder (tree):
    if tree:
        print tree.getRootVal()
        preorder (tree.getLeftChild())
        preorder (tree.getRightChild())
```

Gerçekleme_43

```
def preorder (self):
    print self.key
    if self.left:
        self.left.preorder()
    if self.right:
        self.right.preorder
```

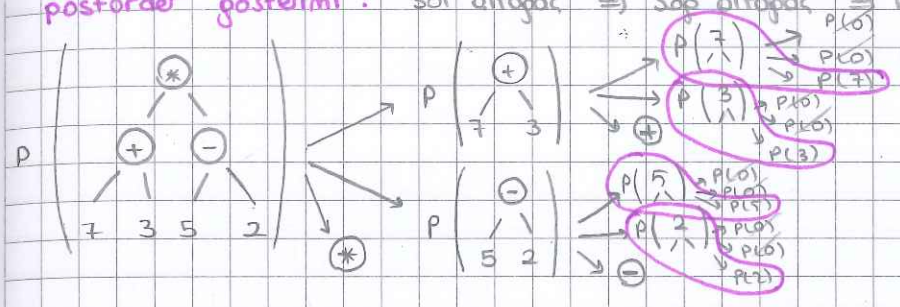
inorder gösterim: sol altağaç ⇒ kök ⇒ sağ altağaç



Gerçekleme_46

```
def inorder (tree):
    if tree != None
        inorder (tree.getLeftChild())
        print tree.getRootVal()
        inorder (tree.getRightChild())
```

postorder gösterimi : sol altağaç ⇒ sağ altağaç ⇒ kök



Gerçekleme - 44

```
def postorder(tree):
    if tree!=None:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print tree.getRootVal()
```