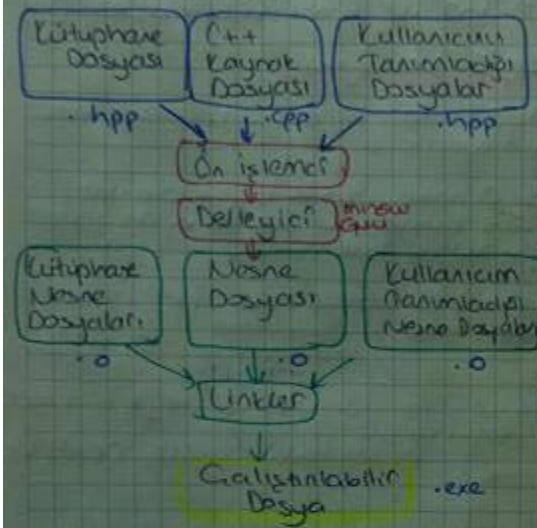
VERİ YAPILARI 2015-2016 VİZE ÇALIŞMALARI ①

Bölüm-1-Giriş

#include <iostream>

Ön işlemci direktifi



C++ programının giriş yeri (başlangıç noktası) MAIN fonksiyonudur.

MAIN → parametreli ? kullanılabilir.
parametresiz ?

using namespace std; → alan uzayı

Tüm arası dönüşümde, büyük türden küçük türe dönüşümde (Unpoint) VERİ KAYBI olur

Sınıf ve Struct arasında küçük bir fark vardır.

→ SINIF (CLASS) elemanları varsayılan olarak private erişimlidir

→ STRUCT (YAPI) elemanları varsayılan olarak public erişimlidir

public : Sınıf dışından, sınıf nesnesi oluşturulmadan erişilebilir.

private : Sınıf dışından hiçbir şekilde erişilemez

protected : Sadece sınıf içinden ve sınıfı kalıtlayan diğer sınıfların erişimi var.

SINIF METODLARI

① YAPICI METODLAR

Bir sınıftan nesne oluşturulacağı zaman ilk çağrılan metod. Birden çok yapıcı metod olabilir.

② YIKICI METODLAR

Bir sınıftan oluşturulan nesnelerin bellekte geri alınması için çağrılır. 1 sınıfta 1 tane bulunur.

* Bir sınıf birden çok şekilde oluşturulabilir; ama SADECE TEK 1 şekilde yıkılabilir.

Nesne Yönelimli Programlama Prensipleri

① Soyutlama → gerçek varlığın basitleştirilmesi
② Kapsülleme → bilgileri saklayıp sınırlı erişim
③ Kalıtım → var olanları kullanıp yeni özellik yazma
④ Çok Biçimlilik

ARAYÜZLER

Arayüzde → metod imzaları (olayların) bulunur.

Sınıf → metod içeriği (olayların) bulunur.

Arayüzden nesne türetilmez!
Sınıftan nesne türetilir.

C++'da arayüz tanımı için
Virtual metotlar kullanılır.

ÖR.

```
class ISekil
{
public:
    virtual int Alan()=0;
};

class Dikdortgen: public ISekil
{
private:
    int genislik, yukseklik;
public:
    Dikdortgen(int en, int boy)
    {
        genislik=en;
        yukseklik=boy;
    }
    virtual int Alan()
    {
        return genislik*yukseklik;
    }
};
```

ISekil arayüzden kalıtım yapılıyor

Algoritma Karmaşıklığı

Belli sayıdaki elemanı işlemden geçirmek için atılan adımsayısı algoritma karmaşıklığı hakkında bilgi verir.

Eleman sayısı (n)
Algoritma karmaşıklığı O(.)

n=100 ise algoritma karmaşıklığı O(n)

Big-O Notasyonu

Algoritmaları birbiriyle karşılaştırmak için kullanırız.

Big-O notasyonunda, büyük oranlar yani tahmini bir şekilde artan sonsuz durumda bulundurur.

1 için 2 for döngüsünde işlem kadar döngü çalışır. Algoritma karmaşıklığı $O(n^2)$'dir.

1 için 3 for döngüsünde işlem kadar döngü çalışır.
Algoritma karmaşıklığı $O(n^3)$'tür.

Çalışma Zamanı Analizi:

Bir algoritmanın giriş veri sayısı(N) sonsuza giderken, algoritmanın yapmış olduğu temel işlem sayısının nasıl değiştiğini ifade eden fonksiyondur.

Big O Notasyonu

$T(N) = O(f(n))$

$O(n) \rightarrow$ fonksiyon değil!
$f(n) \rightarrow$ fonksiyon

ÖR. $3n^2+2n+5 = O(n^2)$

↓
karesel zaman karmaşıklığı

O(1) SABİT ZAMAN KARMAŞIKLIĞI

Girdinin miktarından bağımsız olarak bir yada sabit bir sayıda komutun çalıştığı algoritmalar. max. değere sabittir.

ÖR: n elemanlı dizinin i'inci elemanına değer atamak istiyoruz. Bu değerin atanması o(1)'dir. Çünkü bir elemana indisinden doğrudan erişilir.

B + (i - 1) * C

O(n) DOĞRUSAL ZAMAN:

Giriş sayısıyla işlem arasında bir oran var.

ÖR: Sayı dizisindeki en büyük sayıyı bulan algoritma. Karşılaştırma yapılıyor. Eleman sayısı arttıkça karmaşıklıkta artar.

O(log2 n) LOGARİTMİK ZAMAN:

Eleman sayısı artıyor, ama yaptığımız işlem sayısı eleman sayısıyla orantılı değil.

ÖR: İkili arama ağacı yapısı, Quick sort

O(n²) KARESEL ZAMAN

Bubble sort
Selection sort

O(nlogen)

Problemin küçük parçalara bölünüp çalışması.

Hızlı Sıralama Algoritması
merge sort

O(n³) KÜBİK ZAMAN

İç içe döngüler, sayaçlar (3 tane bir döngüsü)

O(2ⁿ) ÜSTEL ZAMAN

Algoritmada olasılıklar söz konusuysa geriye dönmek gerekir.

* Algoritma zaman karmaşıklığında ulaşmak istediğimiz ideal yakın 1'dir.

#!

O(1) < O(log n) < O(n) < O(n⁹) < O(6ⁿ) < O(n!) < O(nⁿ)

BÖLÜM 2 - VERİ YAPILARI

Verinin ve bilginin bellekte nasıl organize edildiğini, bellekte tutulma biçiminize ve sırasını gösteren yapılar VERİ YAPILARI'dır.

1 bit → 0 } değerlerini alabilir.
→ 1 }

TİP DÖNÜŞÜMÜ

Bir tip dönüşümünde, bir nesne, kendi tipindeki tüm değerleri içermeyen bir tipe dönüştürülüyorsa, bu tip dönüşüme DARALTAN DÖNÜŞÜM denir.

Eğer bir değişken, kendi tipinin tüm değerlerini içeren bir tipe dönüşüyorsa buna GENİŞLEYEN DÖNÜŞÜM denir.

✗ Genişleyen dönüşüm daha güvenlidir.

ÖR: → int → float dönüşümü genişleyen dönüşümdür.

Derleyici tarafından gerçekleştirilen ve zorunlu dönüşüm olarak sağlanan bu tip dönüşüme ÖRTÜLÜ (IMPLICIT) DÖNÜŞÜM denir. Bu tip dönüşümler, derleyicinin tasarımı sırasında belirlenir.

Veri Tipi

İlkel Veri Tipi (int, long, bool...vb)

Kullanıcı Tanımlı Veri Tipi (enum, dizi, record, struct, class)

Bölüm-3 -ÖZYİNELEME

Böl-yönet tekniğiyle algoritma daha küçük parçalalar çalışıyor.

İkili ağaç veri yapısında, arama işlemi gerçekleştiriyoruz. 1 düğümün içeriği, aradığımız eleman düğümden küçükse sola, büyükse sağ tarafta arıyoruz. Bu algoritmayı yazdığımızda, alt düğümlerde algoritma kendini çağırır.

Bu yaklaşıma **REKÜRSİF YAKLAŞIM** denir. Böyle algoritmalara da **REKÜRSİF ALGORİTMALAR** denir.

Rekürsif Yaklaşımda,

1. Tümevarım yöntemi olmalı (Kayısı ağacı örneği)
2. Rekürsif bir algoritma tasarladıysak, alt problemlere bölme var demektir. ve hepsi aynı yöntem ve yöntemlerle çözülecek alt problemlere bölünür.

Orijinal problemin kayısı olan ve bu parçaları çağıran algoritmalara, rekürsif algoritma diyoruz.

Rekürsif Algoritma

- En küçük parçalar (Rekürsiften önce çözülen en küçük parçalar)
- Orijinal problemin aynısı olan ve aynı yöntemle çözülen diğer küçük parçaları çağıran yöntem

Ör Faktöriyel Sayısının Tanımlanması

n sayısının faktöriyeli

$n! = n * (n-1) * (n-2) * \dots * 2 * 1$ ve $0! = 1$

$n! = n * (n-1)!$

Fonksiyon burada kendisini çağırıyor

n=0 ise sonuç 1
n! = 0 ise n * (n-1)!

for $n >= 0$: $n! = \begin{cases} 1 & \text{if } n=0 \\ n(n-1)! & \text{otherwise} \end{cases}$

Program 1:

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

Rekürsif adımlar

3! hesabı

x = factorial(3)

return 3 * factorial(2); 3*2=6
return 2 * factorial(1); 2*1=2
return 1 * factorial(0); 1*1=1
1

Yukarıda yığıt veri yapısı kullanıldı. Alt fonksiyonlar yığıt veri yapısını kullanıyor. İlk çağırılan diğer işlemleri bekletmeli.

İTERATİF YAKLAŞIM ile faktöriyel hesabı (yineleneli)

Ör

```
double factorial (int N) {
    int i;
    double f=1;
    for (i=1; i<=N; i++) f=f*i;
    return f;
}
```

5

1*1=1
1*2=2
2*3=6
6*4=24
24*5=120
} 5!

NOT:

İteratif algoritma → DÖNGÜ YAPISINI

Rekürsif algoritma → DALLANMA YAPISINI kullanır!

→ Rekürsif fonksiyon her çağrıldığında sanki fonk. ilk kez çağrılıyor gibi, bellekten yer tahsisi yapılır. Bu yüzden bellek açısından kullanılmalıdır; ama kullanılacak.

→ Rekürsif yaklaşım, kodlamayı kolay hale getirir. Okunabilirlik yüksek.

→ Rekürsif yaklaşımdaki iteratif yaklaşıma geçiş yapılabilir.

* Fibonacci serisi rekürsif yaklaşımla çözülmelidir.

Fibonacci serisi

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-1)+fib(n-2) & \text{otherwise} \end{cases}$$

fib(2) = fib(0) + fib(1)

Kodu:

```
int fib(int n){
    if (n<=1)
        return n;        → temel nokta
    else
        return (fib(n-1)+(n-2));
                  rekürsif nokta
```

ÖR x = fib(5) için

fib(5) = (fib(4) + fib(3))

fib(4) = (fib(3) + fib(2))

fib(3) = (fib(2) + fib(1))

fib(2) = (fib(1) + fib(0))

fib(1) → çağrıldı; ama hesaplanmadı

x = fib(5); komutu için rekürsiyon ağacı

x = fib(5); komutu çağrıldığında, fib(2) önceden bilinse bile detaylarca çağrılıyor. Bu da zaman karmaşıklığı

Yukarıda;

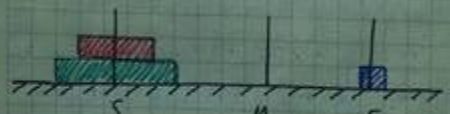
fib(2) → 3 kez çağrıldı
fib(3) → 2 kez çağrıldı
fib(4) → 1 kez çağrıldı

HANOI KULELERİ

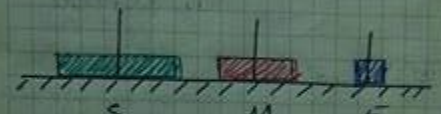
minimum hareket sayısı önemli!

Q1 Üç adet disk ile S çubuğundaki diskleri E çubuğuna aktarın, en büyük disk en altta olacak. (Minumum hareket)

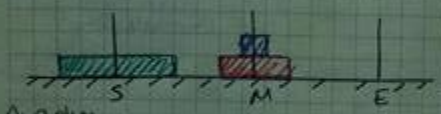
1. adım



2. adım



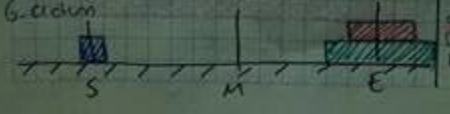
3. adım



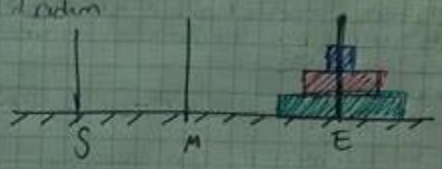
4. adım

5. adım

6. adım



2. adım



Algoritması

Adım 1: Eğer n=1 ise bu diski S'den E'ye taşı

Adım 2: (n-1) adet diski E çubuğunu kullanarak S'den M'ye taşı

Adım 3: Kalan diski S'den E'ye aktar.

Adım 4: (n-1) adet diski M'den E'ye aktar (S çubuğunu kullanarak algoritmanın kendisini çağırdığı kısım)

SIRALAMA VE ARAMA ALGORİTMALARININ KARMAŞIKLIĞI

Sıralama Algoritması	En İyi Durum	En Kötü Durum	Ortalama Durum
Hızlı Sıralama	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$
Yığın, Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log_2 n)$
İkili Arama	$O(1)$	$O(n)$	$O(\log_2 n)$
Sıralı arama	$O(1)$	$O(n)$	$O(n/2)$
Seçmeli sıralama	$O(n^2)$	$O(n^2)$	$O(n^2)$

BÖLÜM 4 BAĞLI LİSTE

Bağlı liste, bilgiye erişim için gerekli adres yada bağlı işaretçi düğümler topluluğudur.

Bu yapıda düğümler fiziksel olarak değil, mantıksal olarak ilişkilidir. Dolayısıyla düğümler, belleğin herhangi bir yerinde olabilirler.

Elimizdeki sıralı diziye yeni bir eleman eklemek için, dizinin "ilk" boş adresine veriyi yerleştirip, diğer elemanlarla karşılaştırılacak. Büyük bir hareket var. Bu da extra zaman maliyeti demek.

Dizi kullanmanın dezavantajları:

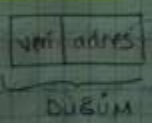
1) Bellek en iyi biçimde kullanılamaz
2) Programın çalışma süresi artar. (Sıralı kalmak zorunda olduğumuz için)
3) Dinamik bellek yönetimi yapılamaz.

* Dizi → statik veri yapısı
  statik bellek yönetimi

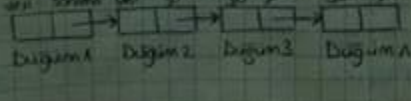
Bağlı liste → dinamik veri yapısı

TEK YÖNLÜ BAĞLI LİSTE

Bir eleman sadece kendinden sonra gelen elemanın adresini tutar. Bir önceki elemanın adresini tutmaz. Buna **TEK YÖNLÜ BAĞLI LİSTE** denir.

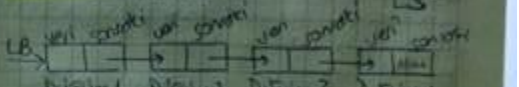


İşaretçi (pointer), yerinin kendisini tutmak yerine adresini tutar. Bu yüzden "sonraki" alanına ihtiyaç duyulur.



Bu yapıdaki bir listenin kendisine ulaşabilmek için özel bir işaretçi kullanılır. Bu işaretçi listenin ilk elemanını gösteren **LB (liste başı)** işaretçisidir.

Bir de listenin sonuna gelip gelmediğimizi kontrol etmek için özel bir bilgiye ihtiyaç duyuyoruz. Bu ise listenin son kısmını gösteren bir özel işaretçi olmalıdır. Ve liste sonunda olacağı için geçersiz adres (NULL) olmalıdır.



LB = NULL ise listede hiç eleman yoktur.
⇒ BOŞ LİSTE

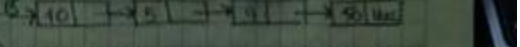
Tek yönlü bağlı liste elemanları:

1) p → bir düğümü işaret eden işaretçi
2) Düğüm(p) → p ile işaret edilen düğüm
3) Veri(p) → p ile gösterilen düğümün bilgi alanındaki veri
4) Sonraki(p) → p ile gösterilen düğümün adres alanındaki değer

BAĞLI LİSTEYE ELEMAN EKLEME

Bir bağlı listeye her an eleman ekleyip her an eleman çıkarabildiğimiz için, bağlı liste dinamik bir veri yapısıdır.

Bağlı listeye eleman eklediğimizde, sadece işaretçi (pointer) güncellemeleri yapılmalıdır.

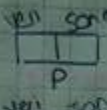


Bu listeye 3 elemanı ekleyelim.

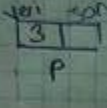
Bunun için önce bir düğüm istediğiz. Bir boş düğüm çağırdık.

Bunu boş düğümler listesinden örneğin getDugum() fonksiyonundan aldığımızı varsayalım.

P=getDugum();

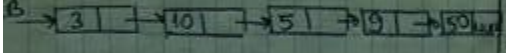


Veri(p)=3;



Yeni düğümü liste başına eklemek istiyorsak, P düğümünün "sonraki" kısmı LB elemanının adresini göstermeli.

sonraki(p)=LB;
LB=P;



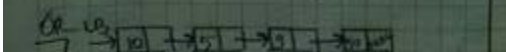
LB ile adresleyen bir bağlı listenin başına eleman eklemek için algoritma:

P=getDugum();
Veri(p)=x;
Sonraki(p)=LB;
LB=P;

BAĞLI LİSTEDEN ELEMAN ÇIKARTMA

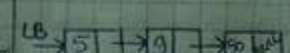
Bağlı listeden eleman çıkarmak için, önce bu bağlı listede eleman olup olmadığı kontrol edilmelidir.

LB=NULL; ise eleman yoktur.



bağlı listenin ilk elemanını çıkartın.

LB artık 10'a değil, LB'nin "sonraki" alanında işaret ettiği düğümü gösterir.



Bunun için yeni bir değişken tanımlamalıyız. ve p=LB ataması yapılmalı.

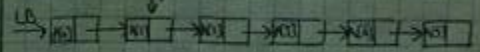
LB=sonraki(p) ifadesiyle LB artık 5'i gösterir.

P=LB; → yeni değişken tanımladık
LB=sonraki(p); → LB 5'i gösterir.
X=Veri(p); → listenin ilk elemanı x'e atandı
Freedugum(p); → p düğümünü boş düğümler listesinde gösterdik

*Bağlı listede bir elemanın işgal ettiği bellek yeri ile onun listedeki pozisyonu arasında bir bağlantı yoktur. Bağlı liste fiziksel sıralı değil, mantıksal sıralı düğümler topluluğudur.

BAĞLI LİSTEDE ARAYA ELEMAN EKLEME

p listenin herhangi bir elemanına işaret etsin. Problemi "herhangi bir x elemanının p düğümünden sonra ekle" olarak belirleyelim.

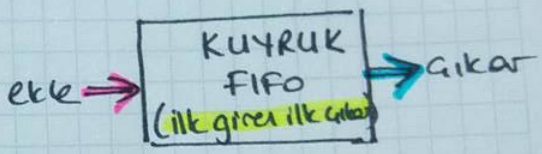


q=getDugum()
Veri(q)=x;
q)=sonraki(p);
p)=q;

YIĞIT VE KUYRUK VERİ YAPISI KARŞILAŞTIRMA

YIĞIT / YIĞIN (STACK)

KUYRUK



İŞLEM	YIĞIN
Ekleme	push
Çıkarma	pop
Sıfırlama	reset

İŞLEM	KUYRUK
Ekleme	add
Çıkarma	get
Sıfırlama	reset

infix ifade postfix ifadeye çevirilirken, İŞLEM ÖNCELİĞİNE bakmak gerekir.

İŞLEM ÖNCELİĞİ (BÜYÜKTEN KÜÇÜĞE)

1. Üs alma (^)
2. Çarpma (*) Bölme (/) → Aynı önceliğe sahip
3. Toplama / Çıkarma → Aynı önceliğe sahip

NOT: Parantezsiz ve aynı önceliğe sahip işlemcilerde işlemler soldan sağa doğru yapılır. (Üs alma hariç!)
Üs almada sağdan sola doğrudur.
Parantezler default öncelikleri belirtmek için konulmuştur.

ADIM 1: İfadeyi soldan sağa karakter karakter tara.
Operand ise → yaz
Operatör ise → yığıta at

ADIM 2: 1 operatör yığıtta kendisine eşit ya da daha yüksek öncelik düzeyine sahip bir operatör gelinceye kadar yığıtta bekletilir.
Kendisine eşit ya da daha yüksek öncelikli operatör geldiğinde ise yığıttan çıkarılır → yazılır. Diğer operatör yığıta atılır.

ADIM 3: Okunacak simge kalmadığında, yığıtta bekletilen operatörler yığıttan çıkartılarak → yazılır.

ADIM 4: DUR!