VERİ YAPILARI DERS NOTLARI

HAZIRLAYAN

Doç.Dr. Abdullah ÇAVUŞOĞLU

ANKARA - 2000

İÇİNDEKİLER

1. PROGRAMLAMA PRENSİPLERİ

1.1 Giriş

Büyük bilgisayar programları yazarken karşılaştığımız en büyük sorun programın hedeflerini tayin etmek değildir. Hatta, programı geliştirme aşamasında hangi metotları kullanacağımız hususu da altından kalkılamayacak bir problem değildir. Örneğin bir kurumun müdürü *"tüm demirbaşlarımızı tutacak, muhasebemizi yapacak kişisel bilgilere erişim sağlayacak ve bunlarla ilgili düzenlemeleri yapabilecek bir programımız olsun"* diyebilir. Programları yazan programcı bu işlemlerin pratikte nasıl yapıldığını tespit ederek yine benzer bir yaklaşımla programlamaya geçebilir (bu genelde uygulanan metottur).

Bu yaklaşım çoğu zaman başarısız sonuçlara gebedir. Programcı işi yapan şahıstan aldığı bilgiye göre programa başlar ve ardından yapılan işin programa dökülmesinin çok kolay olduğunu fark eder fakat, söz konusu bilgilerin geliştirilmekte olan programın başka bölümleri ile ilişkilendirilmesi söz konusu olunca işler biraz daha karmaşıklaşır fakat biraz şans biraz da programcının kişisel mahareti ile sonuçta ortaya çalışan bir program çıkarılabilir. Fakat bir program yazıldıktan sonra genelde bazen küçük bazen de köklü değişikliklerin yapılmasını gerektirebilir. Esas problemler de burada başlar. Zayıf bir şekilde birbirine bağlanmış program öğeleri bu aşamada dağılıp iş göremez hale çok kolaylıkla gelebilir.

Bu kitabın ana amacı programlama metotlarını etkin bir şekilde ortaya koyarak kullanıcılarının boyutları gereğinden büyük olmayan, hızlı çalışabilen, mantıksal yaklaşımları üzerinde toplayan unsurları programcıya iyi bir dil ile aktarmaktır. Bu aşamada en karşımızdaki en büyük engel problemin ne olduğuna karar vermek, anlamsız ve çelişkili isteklerin nasıl bertaraf edileceği hususunda kullanıcıyı bilgilendirmek ve kullanıcıyı program geliştirmeye yönelik olarak uygun metot ve prensibler konusunda bilgi sahibi kılmaktır.

Bilgisayar ile ilgili işlerde en başta aklımıza bellek gelir. Bilgisayar programları gerek kendi kodlarını saklamak için veya gerekse kullandıkları verileri saklamak için genelde belleği saklama ortamı olarak seçerler. Bunlara örnek olarak karşılaştırma, arama, yeni veri ekleme,

silme gibi işlemleri verebiliriz ki bunlar ağırlıklı olarak bellekte gerçekleştirilirler. Bu işlemlerin direk olarak sabit disk üzerinde gerçekleştirilmesi yönünde geliştirilen algoritmalar da vardır fakat bunlar bu kitabın konuları içerisinde yer almayacaktır. Yukarıda sayılan işlemler için bellekte bir yer ayrılır. Aslında bellekte her değişken için yer ayrılmıştır. Örneğin Pascal dilinde tanımladığımız bir *x* değişkeni için bellekte bir yer ayrılmıştır. Bu *x* değişkeninin adresini tutan başka bir değişken de olabilir. Buna pointer denir. Pointer içinde bir değişkenin adresini tutar.

Bilgisayar belleği programlar tarafından iki türlü kullanılır:

- Statik programlama
- Dinamik programlama

Statik programlamada veriler programların başlangıcında sayıları ve boyutları genelde önceden belli olan unsurlardır örneğin:

```
VAR
    x: integer
    y: real
Begin
    X := 3;
    Y := 3.141 ;
End.
```

Şeklinde tanımladığımız iki veri için Pascal derleyicisi programın başlangıcından sonuna kadar tutulmak kaydı ile bilgisayar belleğinden sözkonusu verilerin boyutlarına uygun bellek yeri ayırır. Bu bellek yerleri programın yürütülmesi esnasında küçük program örneğinde de görüldüğü gibi her seferinde x ve y değerlerinde yapılacak olan değişiklikleri kaydederek içinde tutar. Bu bellek yerleri program boyunca statik'tir. Başka bir deyişle programın sonuna kadar bu iki veri için tahsis edilmişlerdir ve başka bir işlem için kullanılamazlar.

Dinamik programlama esas olarak yukarıdaki çalışma mekanizmasından oldukça farklı bir durum arz eder. Yine Pascal programlama dilinden bir örnek verecek olursak:

```
VAR
    IntPointer      : ^ Integer ;
    StringPointer  : ^String ;
Begin
    New(IntPointer) ;
    New(StringPointer) ;
        .
        .
end.
```

Yukarıdaki küçük programda tanımlanan *IntPointer ve StringPointer* işaretçi değişkenleri için New procedure'ü çağrıldığı zaman *IntPointer* için 2 Byte'lık ve *StringPointer* için 256 byte 'lık bir bellek alanını Heap adını verdiğimiz bir nevi serbest kullanılabilir ve programların dinamik kullanımı için tahsis edilmiş olan bellek alanından ayırır. Programdan da anlaşılabileceği üzere bu değişkenler için başlangıçta herhangi bir bellek yeri ayrılması söz konusu değildir. Bu komutlar bir öngü içerisine yerleştirildiği zaman her seferinde söz konusu alan kadar bir bellek alanını heap'den alırlar. Dolayısıyla bir programın heap alanından başlangıçta ne kadar bellek isteminde bulunacağı belli değildir. Dolayısı ile programın yürütülmesi esnasında bellekten yer alınması ve geri iade edilmesi söz konusu olduğundan buradaki işlemler dinamik yer ayrılması olarak adlandırılır. Kullanılması sona eren bellek yerleri ise:

```
Dispose(IntPointer) ;
Dispose(StringPointer) ;
```

Komutları ile iade edilir. Söz konusu değişkenler ile işimiz bittiği zaman mutlaka Dispose procedure'ü ile bunları Heap alanına geri iade etmemiz gerekir. Çünkü belli bir süre sonunda sınırlı heap bellek alanının tükenmesi ile, program *"out of memory"* veya *"out of heap"* türünden bir hata verebilir.

Konuyu biraz daha detaylandırmak için bir örnek verelim: bir stok programı yaptığımızı farz edelim ve stoktaki ürünler hakkında bazı bilgileri (kategorisi, ürün adı, miktarı, birim fiyatı vs.) girelim. Bu veriler tür itibarı ile tek bir değişken ile tanımlanamazlar yani bunları sadece bir tamsayı veya reel değişken ile tanımlayamayız çünkü bu tür veriler genelde birkaç türden değişken içeren karmaşık yapı tanımlamalarını gerektirirler. Dolayısıyla biz sözkonusu farklı değişken türlerini içeren bir RECORD yapısı tanımlarız.

Değişik derleyiciler değişik verilerin temsili için bellekte farklı boyutlarda yer ayırma yolunu seçerler. Örneğin bir derleyici bir tamsayının tutulması için bellekten 2 Byte'lık bir alan ayırırken diğer bir derleyici 4 Byte ayırabilir. Haliyle bellekte temsil edilebilen en büyük tamsayının sınırları bu derleyiciler arasında farklılık arz edecektir.

Yukarıda sözü edilen RECORD tanımlaması derleyicinin tasarlanması esnasındaki tanımlamalara bağlı olarak bellekten ilgili değişkenlerin boyutlarına uygun büyüklükte bir bloğu ayırma yoluna gider. Dolayısıyla stoktaki ürüne ait her bir veri girişinde bellekten bir blokluk yer istiyoruz. Böylece bellek nerede boşluk varsa oradan bize 1 blokluk yer veriliyor. Biz de hem hızı arttırmak hem de işimizi kolaylaştırmak için her bloğun sonuna bir sonraki bloğun adresini tutan bir işaretçi yerleştiriyoruz. Daha sonra bu bellek yerine ihtiyacımız kalmadığı zaman örneğin staktaki o mala ait bilgileri sildiğimiz kullandığımız hafıza alanlarını iade ediyoruz. Ayrıca bellek iki boyutlu değil doğrusaldır. Sıra sıra hücrelere bilgi saklanır. Belleği etkin şekilde kullanmak için veri yapılarından yararlanmak gerekmektedir. Bu sayede daha hızlı ve belleği daha iyi kullanabilen programlar ortaya çıkmaktadır.

2. BAĞLI LİSTELER (LİNKED LİST)

- Tek Bağlı listeler
    - Tek Bağlı Doğrusal listeler
    - Tek Bağlı Dairesel listeler
- İki Bağlı listeler
    - İk Bağlı Doğrusal listeler
    - İki Bağlı Dairesel listeler

‘Liste (list)’ sözcüğü aralarında bir biçimde öncelik-sonralık ya da altlık-üstlük ilişkisi bulunan veri öğeleri arasında kurulur. ‘Doğrusal Liste (Linear List)’ yapısı yalnızca öncelik-sonralık ilişkisini yansıtabilecek yapıdadır. Liste yapısı daha karmaşık gösterimlere imkan sağlar.

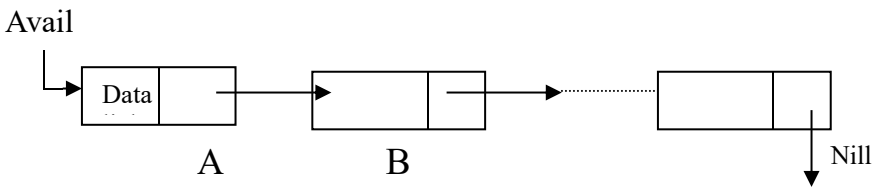
2.1 TEK BAĞLI DOĞRUSAL LİSTELER (One Way Linked List)

Tek bağlı doğrusal dizi, öğelerinin arasındaki ilişki (Logical Connection)’ye göre bir sonraki öğenin bellekte yerleştiği yerin (Memory Location) bir gösterge ile gösterildiği yapıdır.

Bilgisayar belleği doğrusaldır. Bilgiler sıra sıra hücrelere saklanır. Her bir bilgiye daha kolay ulaşmak için bunlara numara verilir ve her birine “node” adı verilir.

Data alanı, numarası verilen node’da tutulacak bilgiyi ifade eder. Link alanı ise bir node’dan sonra hangi node gelecekse o node’un numarası tutulur.

- Link alanı integer türünden bir değer alır.
- Data alanı bölgesindeki numaraları temsil eder.
- Avail: kullanıma hazır, boşta bulunan nodların topluluğunu ifade eder. Avail 1 denilirse bu ifade 1 numaralı ndeyi işaret etmektedir.



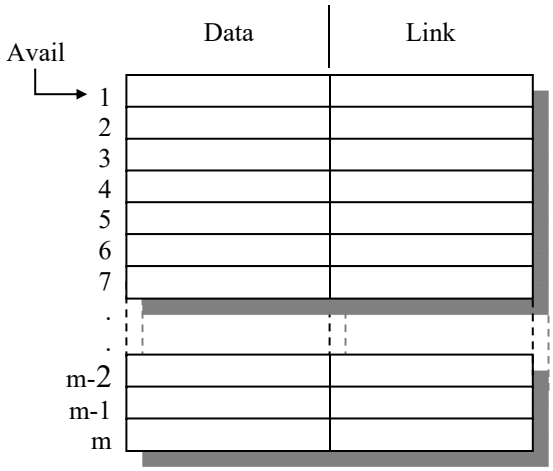
Diyelim ki A, B, C, D gibi bir bloğa ihtiyacımız oldu. Pascal'da uygun komutları kullanarak bu verilerin aralarında bağlantı kurabiliriz. Bu iş için *Şekil* 'deki gibi bir Model Hafıza Alanımız olsun. Yukarıdaki gibi bir Node Uzayını tanımlamak için kullanacağımız kod aşağıdaki gibi olacaktır.

Bir bağlı doğrusal listelerde Node Uzayı (Node Space)'nı tanımlayan program.

Unit Node_uz

```
Unit Node_uz;
      Interface

           Const
             Memory = 10;
             Nil = 0;
            Var
               Avail: integer;
               Node: array[1..memory] of  record
                                             Data: real;
                                             Link: integer;
                                          End;
      Implementation
End.
```

Bu altyordamda 10 adet node alanı ayırıyoruz. Avail node uzayında boş kullanılabilir alanları gösterir. Unit Node_uz unitinde avail'i integer, node'u ise record tipli dizi değişken olarak tanımlıyoruz. Node uzayımız 10 adet hafıza alanından oluştuğundan dolayı sabit tanımlama bloğunda memory'yi 10 eşitledik. Eğer daha büyük hafıza alanı gerekirse sabit tanımlama bloğunda memory'nin değerini arttırabiliriz. Link dediğimiz kısım, bir sonraki node'u gösteren pointer'dır. Integer tipi olması yeterlidir. Data kısmını ise real tipinde bilgi saklayacağımızdan dolayı real olarak tanımladık.



Bu tanımlamalardan sonra Şekil'deki gibi bir Node Uzayı'na sahip oluruz.

Bir bağlı listelerde memoryde node,link ve data alanı oluşturulup bir bağlı listelerde nodeları başlangıç durumuna getiren program.

Unit AltYor0

```
Unit   AltYor0;
   interface (**)
         procedure   dumpmem;
         procedure   nodeinit;
   implementation  (**)

   uses Node_uz;

procedure   dumpmem;
   var
    i:integer;

 begin
    for i:=1 to  memory do
    writeln ('Node[',i:3,']=',node[i].data,',',node[i].link);
 end;

procedure  nodeinit;
   var
    i:integer;
   begin
    Avail:=1;
    for  i:=1 to  memory-1 do
    node[i].link:=i+1;
    node[memory].link:=nill;
   end;
```

```
end.
```

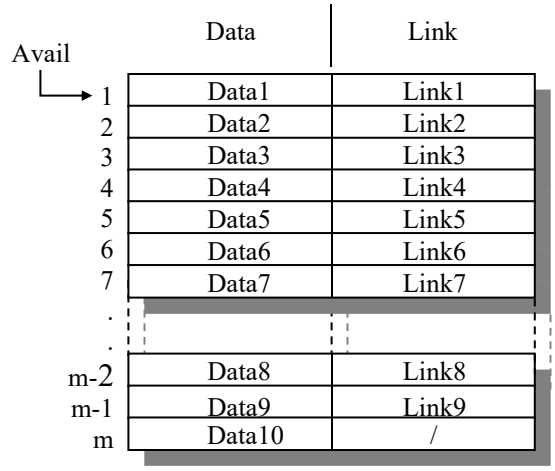
Fonksiyonlardaki değişkenlerin ömrü fonksiyon sonuna kadar geçerlidir. Fonksiyon bitince hafızada yer kaplamaz. Avail; kullanılmayan node'ları gösterir. Fonksiyon başına VAR yazılmazsa program boyunca yapılan değişiklikler List dizisini etkilemez. Dinamik değişkenler de yer sadece gerektiğinde istenir.

Function ve procedure tanımlama tipleri

- Call pass by Value
- Call Pass by Reference
- Call Pass by Name
- Call by result

procedure dumpmem : Bu procedure node uzayımızdaki node'ları hem data kısmını hem de linklerini ekrana basıyor.

procedure nodeinit : Bu procedure linklerin birbirine artarda bağlanmasını sağlar. Burada her link bir sonraki node'u gösterecek şekilde tanımlanıyor (node[i].link :=i+1). Bir bağlı doğrusal dizi kullanacağımızdan dolayı en son node'un linkini daha önceden Node_uz unitinde 0'a eşitlediğimiz nill'e eşitledik (node.[memory].link :=nill).



Bu procedure çalıştıktan sonra node uzayımız aşağıdaki şekil gibi olur.

Unit AltYor1

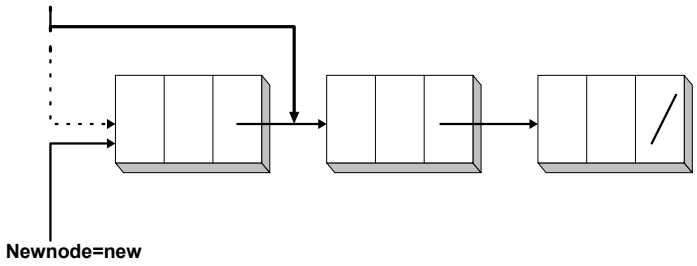
Aşağıda yazılan unit ve ilk tanımlanan fonksiyon olan newnode fonksiyonu ile kullanılmak üzere node uzayından bir node koparılır, koparılan bu node avail olarak tanımlıdır kullanılmak için hazır beklemektedir, koparılan bu yeni node'dan sonra node uzayındaki avail diğer hazır bekleyen node'a eşitlenir.

```
unit  AltYor1;
    interface(**)
    function  newnode:integer;
    function  cuthead(var  list:integer):integer;
    function  last(list:integer):integer;
    procedure  concatenate (var L1:integer; L2:integer);
    procedure  free(list:integer);

    implementation(**)

    uses   node_uz, altyor0;

  function  newnode :integer ;
     var
       new:integer;
  begin
     new:=cuthead(avail);
       if  new=nill
            then begin
                 writeln('nodespace full...') ;
                 halt;
               end;
               {else}
               newnode:=new;
  end;

  function cuthead(var list:integer):integer;
  begin
     cuthead:=list;
       if  list< >nill then
              list:=node[list].link;
  end;


  function last({the value of}list:integer):integer;(****)
  begin
     if list< >nill  then
          while  node[list].link< >nill  do
```

```
            list:=node[list].link;
    last:=list;
  end;

  procedure   concatenate(var L1:integer;L2:integer);
  begin
    if  L1=nill  then  L1:=L2
             else  node[Last(L1)].Link:=L2;
  end;

  procedure  free(list:integer);
  begin
       concatenate(list,Avail);
       Avail:=list;
  end;
end.
```

function newnode : Bellekten gerekli yeri (belli bir değişken için) almak için kullanılır. Bu fonksiyonla node uzayından kullanılmak üzere bir tane node koparılır. Koparılan bu node, node uzayında kullanılmak üzere hazır bekleyen node'dur. Yani avail'dir. Kullanılmak için kullanılan bu yeni node 'dan sonra node uzayındaki avail diğer hazır bekleyen node'ye eşitlenir.

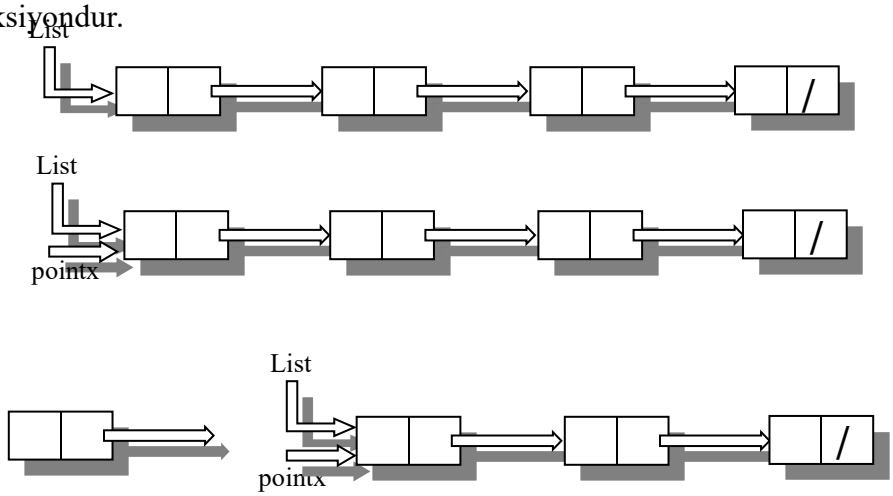
Cuthead fonksiyonu, list isimli bir değişken pas ediliyor (Aslında bu list bir dizi) Cuthead =list diyoruz. Eğer bu list 0 elemanlı bir dizi ise cuthead fonksiyonu geriye 0 döndürecektir. Cuthead ile dizinin ilk node'unu kes gönder diyoruz. Yani; list 0 değilse; 1. node cuthead'in içine alınır. List de ilk node'a eşitken ikinci node'a eşit hale gelir.



newnode function'u çalıştırılınca yukarıdaki gibi yapı gerçekleşir.

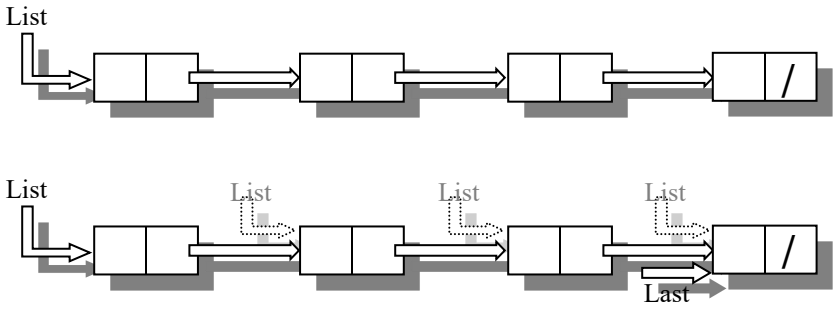
Avail'in 0 olması demek, kullanıma hazır node yok demektir. Bu durumda "Node Space Full" şeklinde bir mesajla karşılaşırız.

function cuthead : Biraz önce bahsedilen istenmeyen kutucuğun avail dizisine atılmasını sağlayan fonksiyondur.



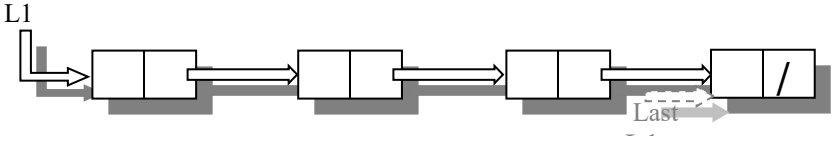
Function cuthead çalışınca yukarıdaki şekildeki yapı meydana gelir.

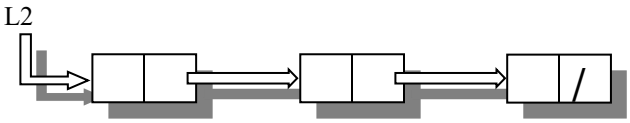
function last : Bu fonksiyonla sonuncu node bulunur. Öncelikle list diye bir şeyin nill olmaması ve linkinin nill olmaması şartları kullanılarak sonuncu node bulunur.



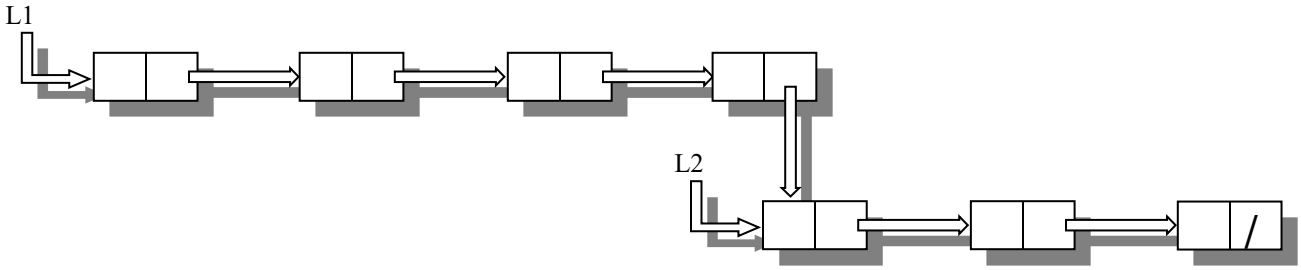
Elimizde bir dizi olsun. Bu dizinin en sonuna elemanından sonra bir eleman daha ekleyin dense last fonksiyonunu kullanabiliriz. Eğer dizi 0 elemanlı ise last'ı yoktur. last=list olur. Eğer dizi 0 elemanlı değilse list'i bir ilerletir. list 0 olana kadar, 0 bulunca dizinin sonu bulunmuş demektir.

procedure concatenate : Diyelim ki elimizde L1 ve L2 listeleri var. Bunları birleştirmek için kullandığımız fonksiyondur.

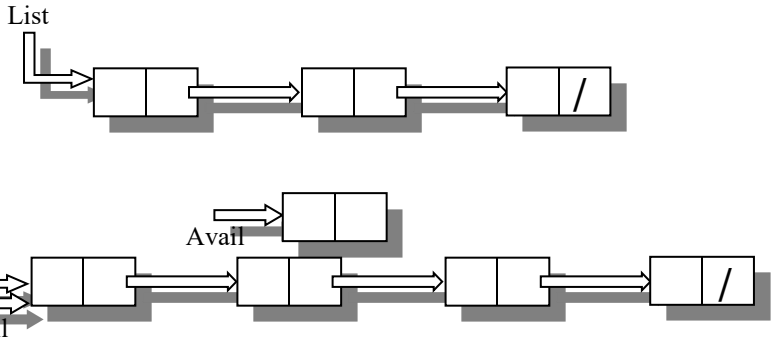




Örneğin L2'yi L1'in arkasına ekliyoruz, eğer L1 boşsa. L1 dolu ise, L1'in en son node'unu buluyoruz. Daha sonra L1'in last'ı L2 olsun diyoruz.



procedure free : Elimizde bir liste var. Bu liste ile işimiz bittiğinde bunların bellekte yer işgal etmemesi ve tekrar kullanılabilir hale getirilmesi gerekir. Bu procedure ile kullanılıp işi biten dizi node uzayına gönderilir. Yani list'i avail dizisinin başına ekler ve avail'i list olarak yeniler.



Concatenate fonksiyonu ile işi biten list dizisini avail dizisinin başına ekliyoruz. Avail=list diyerek işi biten diziyi avail dizisine aktarmış oluyoruz.

Unit AltYor2

```
Unit  AltYor2;

  interface(**)

  procedure  addhead(node_:integer; var list:integer);
  function   cons(data_:real;link_:integer):integer;
  function   copy(list:integer):integer;
```

```
  function   locate(data_,list:integer):integer;

  implementation (**)

  uses node_uz,altyor0,altyor1;

  procedure  addhead(node_:integer;  var  list:integer);
  begin
    node[node_].link:=list;
    list:=node_;
  end;

  function  cons (data_: real; link_: integer): integer;
    var
      cons_:integer;
  begin
    cons_:newnode;
    cons:=cons_;
    node[cons_].data:=data_;
    node[cons_].link:=link_;
  end;

  function  copy(list:integer):integer;
    var
      suret:integer;
  begin
    suret :=nill;
    if list< >nill
           then  repeat
             concatenate(suret,cons(node[list].data,nill));
             list:=node[list].link;
             until  list=nill;
    copy:=suret;
  end;

  function locate(data_:real;list:integer):integer;
  begin
    locate:=nill;
      while  list<>nill  do
        if   node[list].data<>data_
          then  list:=node[list].link
          else  begin
              locate:=list; exit;
            end
  end
end.
```

procedure addhead : Herhangi bir dizinin başına bir node ekler. List dizisi işlem sonunda değişeceğinden var olarak tanımlanmıştır.

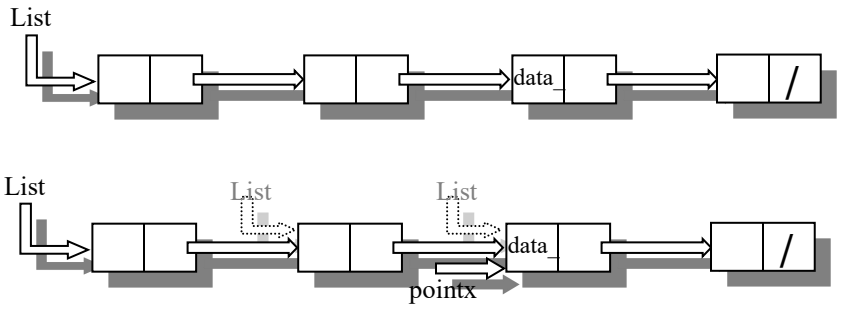
function cons : Node uzayından alınan node'un data ve link alanını doldurmak için bu fonksiyon kullanılır. Yeni bir node alıp, bu node içine yeni değerler yazma işlemini yapar. Bu function ile;

1-)Avail dizisinden yeni bir node alınır.

2-)Bu yeni node'a fonksiyona aktarılan değerler yazılır.

function copy; Bu function herhangi bir dizinin kopyasını alır listin kopyasını oluşturupgeri gönderir. Datalar aynı node nolar farklı olur.

function locate : Bu fonsiyon ile verilen data (data_) 'nın elde bulunan dizide olup olmadığı, eğer varsa data'nın dizinin içerisinde bulunduğu adresi yani numarası bulunur. Liste içinde yer alan bir tane node'u bulmamıza yarar.



Data_'u datalarla karşılaştırıp aynı olanı ararız. En başta "locate :=nill" ile sıfırlıyoruz. İlk node'un data'sı farklı ise list 2. node'u; o da farklı ise list 3. node'u ... göstersin. Aynı datayı bulunca "locate:=list" yapıp çık.

Unit AltYor3

```
unit  altyor3;

  interface(**)

  function  member(node_,list:integer):boolean;
  function  advance(var  point:integer):boolean;
  function  delete(node_:integer; var list:integer):boolean;
```

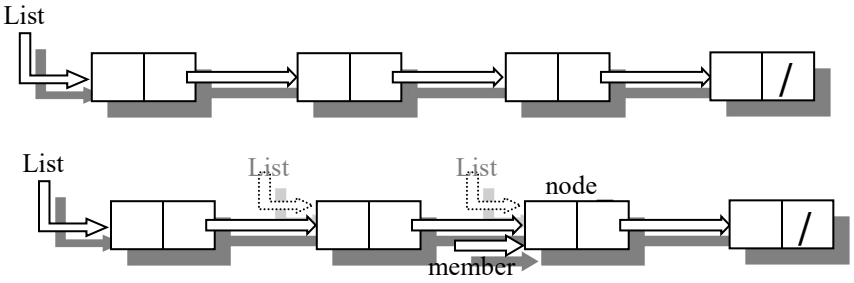
```
  implementation(**)

  uses   node_uz,altyor0,altyor1,altyor2;
  function  member(node_,list:integer):boolean;
  begin
    while  (list  <>  nill) and  (list<>node_)  do
      list:=node[list].link;
    member:=(node_=list);
  end;

  function  advance(var  point:integer):boolean;
  begin
    advance:=false;
    if  point<>nill
           then  if  node[point].link<>nill
                   then
                      begin
                         point:=node[point].link
                         advance:
                      end;
  end;

  function  delete(node_:integer;var  list:integer):boolean;
  var
  point:integer;
  begin
    delete:=false;
    if  list=nill  then  exit;
    if  list=node_
           then  begin
               addhead(cuthead(list),avail);
               delete:=true;
             end;

           else  begin
              point:=list;
              repeat
               if  node[point].link=node_ then
               begin
               addhead(cuthead(node[point].link),avail);
               delete:=true;
               exit;
                     end;
               until  not  advance(point);
             end;
  end;
end.
```

function member : Verilen node bu listede var mı yok mu? Kontrol eden function'dır. list dizinin başını gösteren elemandır.

node_= list ifadesi karşılaştırma ifadesidir.

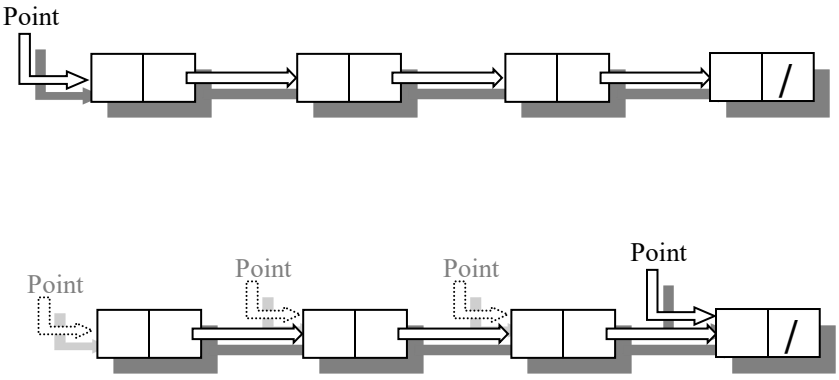
node_= list ise member 1,

node_<>list ise member 0 'dır.



Eğer parametre olarak gönderilen node list'te ise mantıksal true, yoksa mantıksal false değerini gönderir.

function advance : Dizide bir sonraki node'a atlamamızı sağlayan function'dır. Eğer atlama gerçekleşirse mantıksal true, gerçekleşmezse mantıksal false değerini gönderir. Bir dizide elimizdeki pointer'ın gösterdiği node'dan bir sonrakini gösterecek şekilde ilerletir. pointer illa o listenin başını gösterecek diye bir şart yoktur. Eğer point <> nill ise geriye false döner. point dizi üzerinde nerede olduğumuzu gösteriyor. point'in gösterdiği node'un linki nill'e eşit ise bir sonraki node'a atlar.



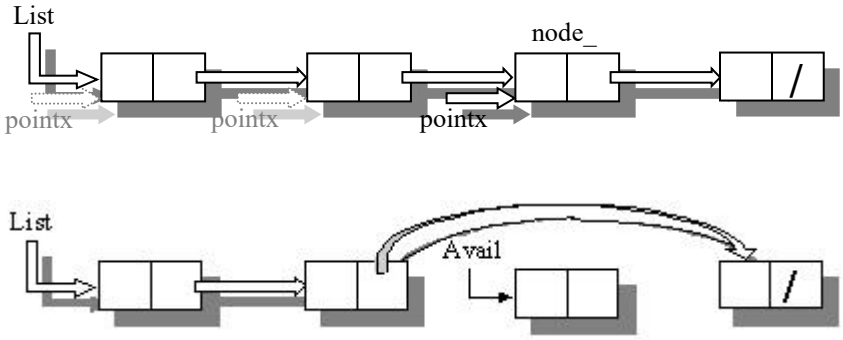
function delete : Dizimizden bir tane node silmek için geliştirilmiş bir function'dır.

list : silinmek istenen node'un bulunduğu liste;

node_silinmesi istenen node'un numarası.

Eğer dizimiz boş ise hemen function'dan çıkılır ve geriye false değeri döndürür.

Cuthead function'u ile listenin ilk node'unu kesip addhead function'u ile avail'in baş tarafına ekleriz. Delete node_'u listeden silip, avail'in baş tarafına ekler. Eğer silinen liste avail ise, node avail listesinden silinir ve availin başına eklenir. Yani avail o node'dan başlar. Bu işlem advance function'u ile ilerleyemeyinceye kadar devam eder.



Şekil 1.4'de verilen listemizden(Şekil 1.4-a) 3 numaralı node'u silmek istesek Şekil 1.4-b'deki liste elde edilir. Cuthead list diyerek dizinin 1.node'unu diziden koparıyoruz. Addhead ile bunu avail dizisinin başına ekliyoruz. 3.node'un link'i 5'i gösteriyor (1.durum)-(silinecek eleman dizinin başındaysa)

Eğer eleman dizinin herhangi bir yerindeyse, önce arama işlemini yapıyoruz. node_u bulduktan sonra cuthead, node[point].link diyoruz. Cuthead ile o node'u koparıyoruz. Daha sonra addhead ile avail'e ekliyoruz.

Progran Nodeini

Yukarıda tanımlanan tüm procedure ve function'lar aşağıdaki gibi bir programda kullanılabilir. Programun uses kısmına daha önceden yazdığımız procedure ve function'ların bulunduğu unit isimlerini ekliyoruz.

```
program nodeini;
  uses
    node_uz,altyor0,altyor1,altyor2,altyor3;
  var
    i,new:integer;
  begin
    nodeinit;  i:=5;
    writeln('delete:',delete(i,avail));
    writeln('av=',avail);
    writeln('lastav',last(avail));
    writeln('cuth=',cuthead(avail));
    new:=newnode;
    writeln('new=',new,'cut..',cuthead(avail),'ava=',avail);
    dumpmem;
  end.
```

Yukarıdaki program çalıştırılırsa aşağıdaki çıktı elde edilir.

Delete:TRUE

Av=5

Lastav10

Cuth=5

New=1cut..2ava=3

Node[1]= 2

Node[2]= 3

Node[3]= 4

Node[4]= 5

Node[5]= 6

Node[6]= 7

Node[7]= 8

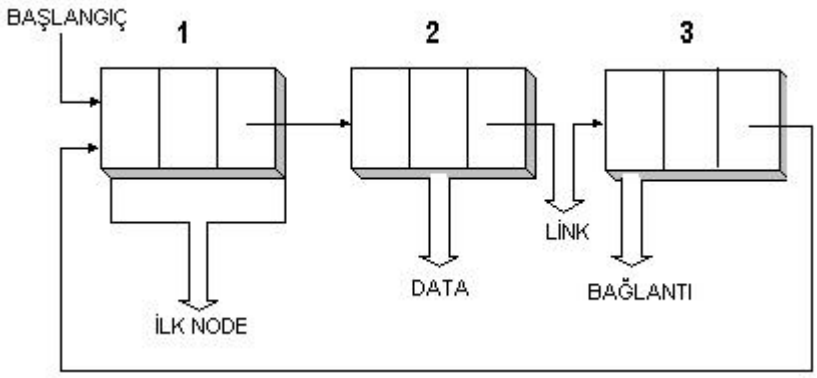
Node[8]= 9

Node[9]= 10

Node[10]= 0

2.2 TEK BAĞLI DAİRESEL LİSTELER

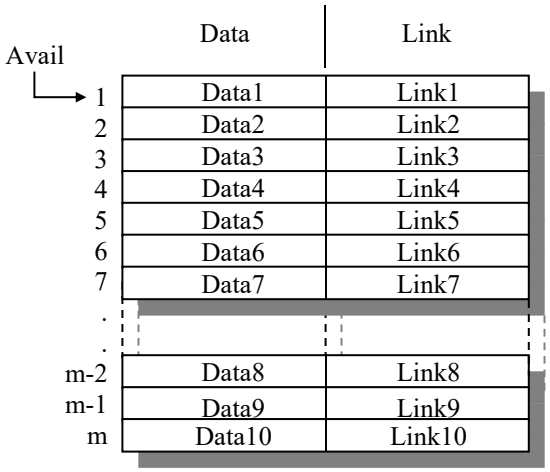
Tek bağlı dairesel listelerde; doğrusal listelerdeki bir çok işlem aynı uygulanır; fakat burada dikkat edilmesi gereken, son node'dan sonra nill değil list olarak belirlenen ilk node'un gelmesidir.



Tek bağlı dairesel listelerde nodeinit, cuthead, last, concatenate, free, addhead, copy, locate ve member function ve procedure'leri değişikliğe uğrar, diğerleri aynı doğrusal listelerdeki gibidir.

Procedure Nodeinit

Bu procedure burada bir bağlı dairesel listelerin node'larını başlangıç durumuna getirir.



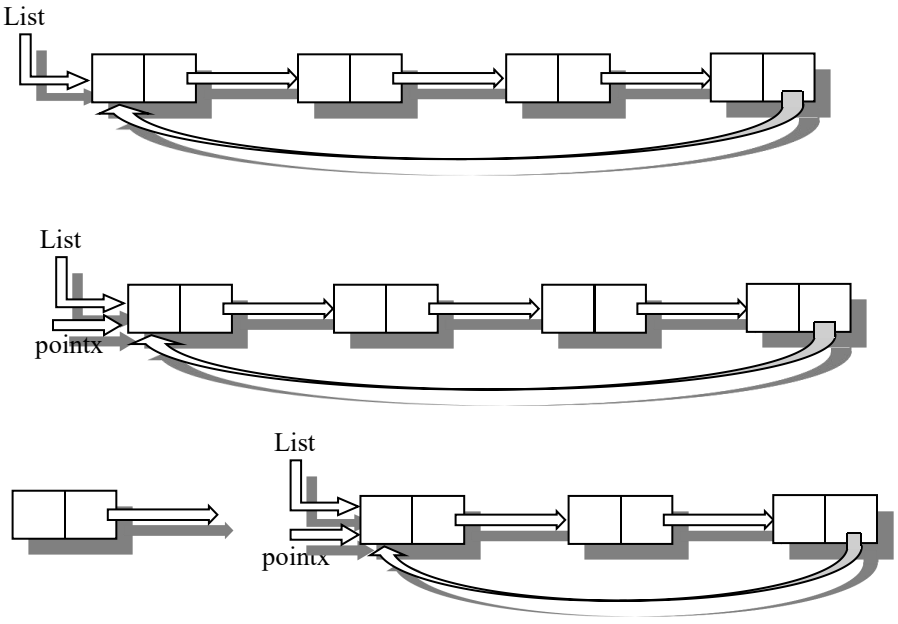

```
Procedure nodeinit;
var
  i:integer;
  begin
    Avail:=1;
```

```
  for i:=1 to memory do
    node[i].link:=i+1;
    node[memory].link:=avail;
  end;
```

Function cuthead

Tek bağlı dairesel listelerde node uzayından ilk node'u koparma.

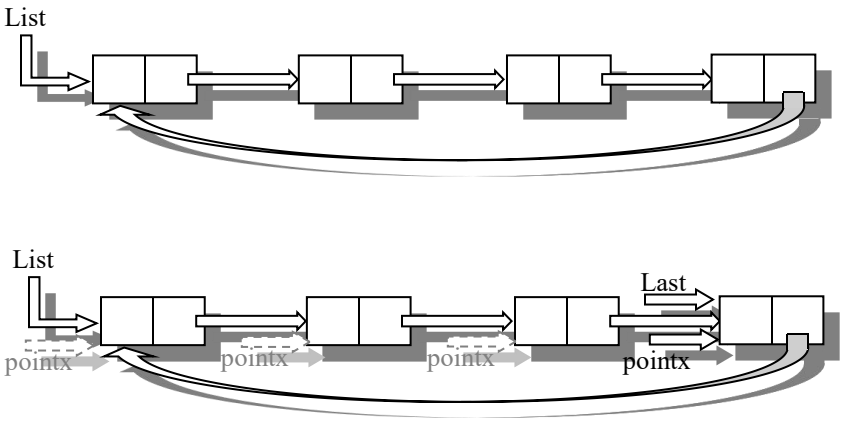
```
Function cuthead(var list:integer):integer;
  var
    Pointx:integer;
  Begin
    Cuthead:=list;
    if list<>nill then
         if node[list].link=list
           then list:=nill
      else  begin
              pointx:=node[list].link
              node[last(list)].link:=pointx;
              list:=pointx;
            end;
  end;
```



Function last

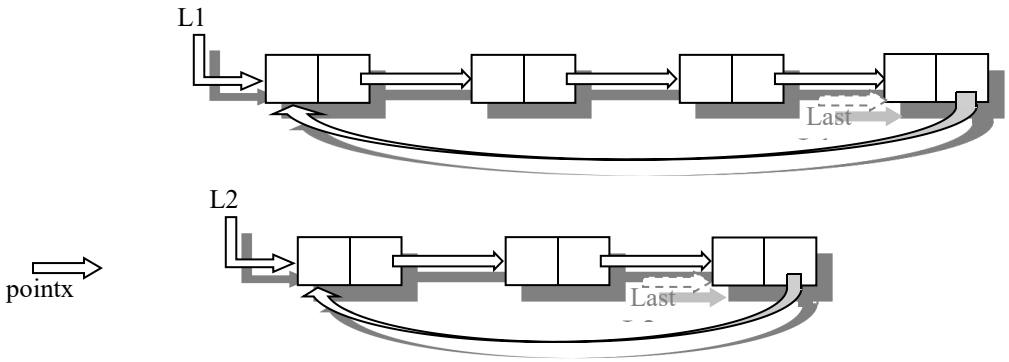
Tek bağlı dairesel listelerde son node'u bulma.

```
Function last(list: integer): integer;
Var
  Point:integer;
  Begin
    Point:=list;
      If  point<> nill then
                    While node[list].link<>list   do
                       Point:= node[point].link;
                          Last:=point;
  End;
```

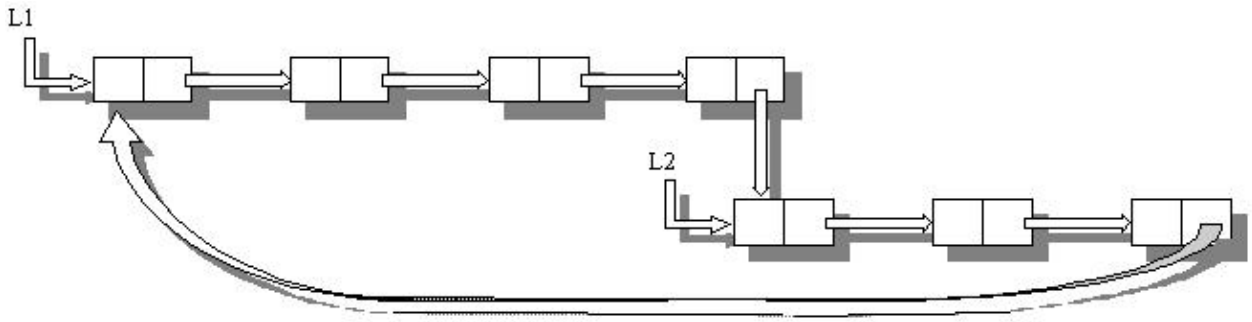


Procedure concatenate

Tek bağlı dairesel listelerde iki diziyi birleştirme. Bu Procedure'ün kullanımında önemli olan doğru işlem sırasını takip etmektir eğer öncelikli yapılması gereken bağlantılar sonraya bırakılırsa son node'un bulunmasında sorunlar çıkacaktır.

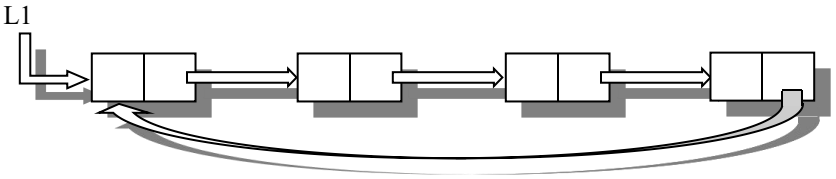


```
Procedure concatenate(var L1:integer;L2:integer);
  Begin
    if L1=nill  then L1:=L2
      else
        begin
          node [last(L1)].Link:=L2;
          node [last(L2)].Link:=L1;
        End;
  End;
```



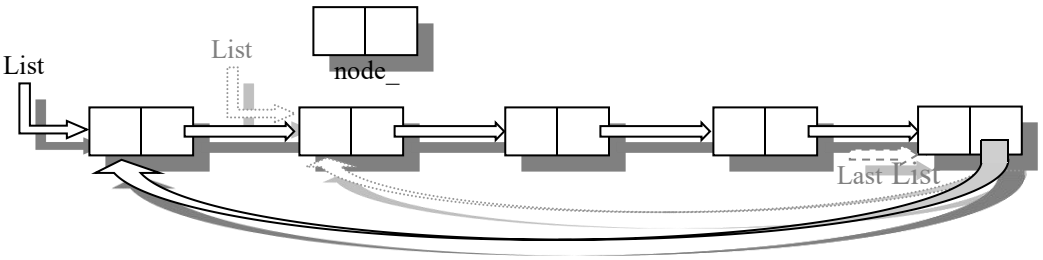
Procedure addhead

Tek bağlı dairesel listelerde node ekleme.



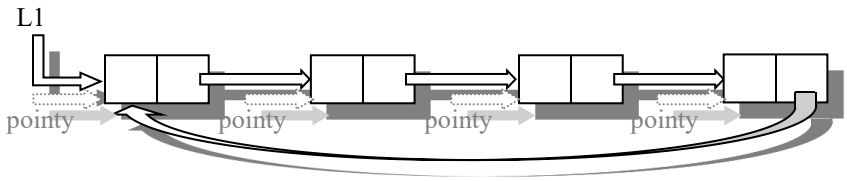

```
Procedure addhead(node_: integer; var list: integer);

  Begin
    If  list<>nill  Then
         Begin
           Node[node_].link:= list;
           List:=node_;
```

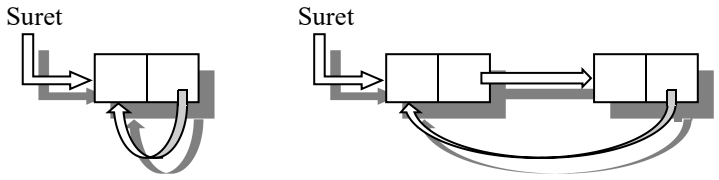
```
          Node[last(list)].link:=node_;
        End;
   Else

        Begin
           List:=node_;
           Node[node_].link:=list;
        End;
End;
```



Function copy

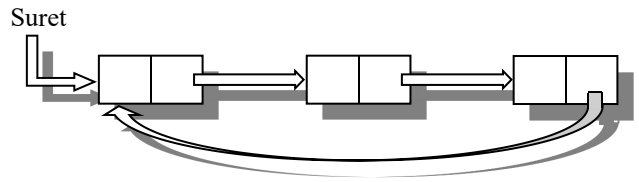
Tek bağlı dairesel listelerde bir dizinin kopyasını elde etme. Copy procedure'ünde cons fonksiyonu yardımıyla avail dizisinden yeni node'lar koparılarak eldeki dizinin node değerleri bu yeni node'lara eşitlenir ve eldeki dizinin kopyası çıkarılmış olur.

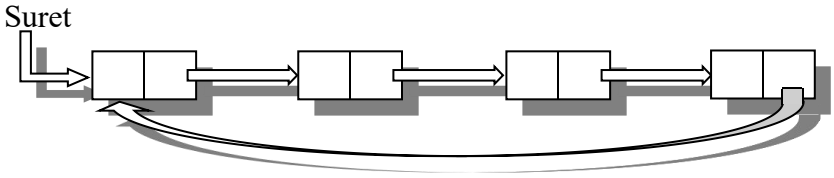





```
Function copy(list: integer): integer;
  Var
    Suret,point,x: integer;
```

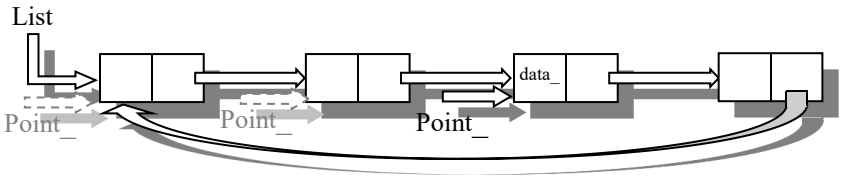
```
Begin
  Suret:= nill; point:=list;
  if  list<>nill  Then
    Repeat
      X:=cons(node[point].data,node[point].link);
      Node[x].link:=x;
      Concatenate(suret,x);
      Point:=node[point].link;
    Until List= nill;
  Copy:= suret;
End;
```





Function Locate

Tek bağlı dairesel listelerde bir node'un yerini bulma. Bu function ile bir node'un yeri bulunup data'sına bakılır.

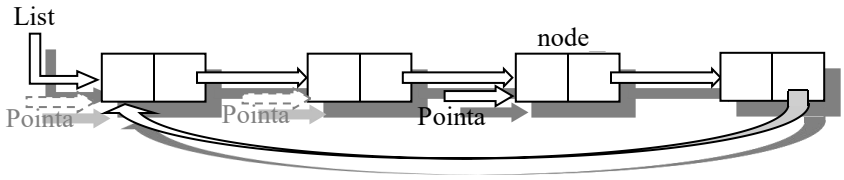



```
Function locate(data_:real, list_: integer): integer;
  Var
    Point:integer;
  Begin
    Locate:= nill;
    Point:=list_;
    If  point<>nill Then
      Repeat
```

```
        If  node[point].data<>data_ Then
          Point:=node[point].link;
        Else
          Begin
            Locate:=Locate+1;
          End;
      Until point=list;
  End;
```

Function member

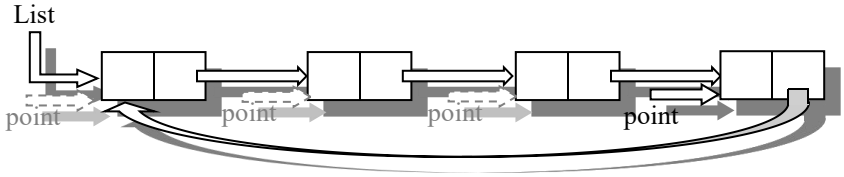
Tek bağlı dairesel listelerde bir node'u arama.




```
Function member(node_, list: integer): boolean;
  Var
    Point:integer;
  Begin
    Point:=list;
    If  point<> nill Then
      Repeat
        If  point<>node_  Then
          Point:= node[point].link;
      Until point=list;
    Member:= node_=list;
  End;
```

Function Advance

Tek bağlı dairesel listelerde bir node kadar ilerleme.

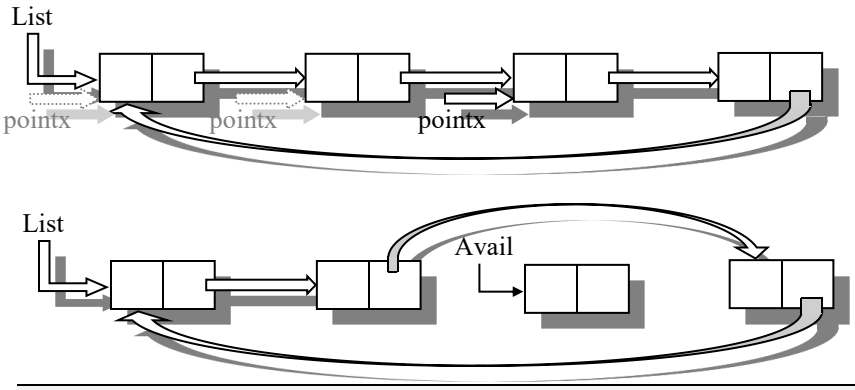



```
Function advance(var point: integer,list:integer): boolean;
```

```
Begin
  Advance:= false;
  If point <> nill Then
       If node[point].link <> nill Then
                  Begin
                     Point:= node[point].link;
                     Advance:= true;
                  End;
End;
```

Function delete

Tek bağlı dairesel listelerde bir node'un silinmesi.




```
Function delete(node_: integer; var list: integer): boolean;

  Var
    Point: integer;
  Begin
    Delete:= false;
    If list= nill Then exit;
    If list= node_ Then
         Begin
           Addhead(cuthead(list), avail);
           Delete:= true;
         End;

        Else
           Begin
             Point:= list;
             Repeat
               If node[point].link= node_ Then
              Begin

               Addhead(cuthead(node[point].link), avail);
               Delete:= true;
               Exit;
```

```
            End;
           Until not advance(point,list);
         End;
  End;

    Else'den itibaren yazılabilecek alternatif program.

         Else
          Begin
            Point:= list;
            While node[point].link<>list do
          Begin
            If node[point].link= node_ Then
            Begin

             Addhead(cuthead(node[point].link), avail);
             Delete:= true;
             Exit;
            End;
            Else point:=node[point].link
          End;
  End;
```

ALIŞTIRMA SORULARI

Soru-1: Bir bağlı doğrusal listeler de function locate (data_: real; list, n: integer):integer; şeklinde yeniden öyle yazınız ki locate'in değeri "data_"nın n. rastladığı node olsun. Eğer verideğeri data_ n. defada bulunamazsa değeri nill olsun.

```
Function locate(data_:real;list,node):integer;
  Var
    a:integer;
  Begin
    a:=0;
    locate:=nill;
    while list<>nill do
      if node[list].data<>data_
        then list:=node[list].link
        else begin
             a:=a+1;
      if a=node then begin
                locate:=list;
                exit;
               end;
      list:=node[list].link;
```

```
          end;
  end;
```

Soru-2: Soru1'i iki bağlı listeler için tekrar çözünüz.

```
Function locate(data_real;list,node:integer):integer;
  Var
    a,point_:integer;
  Begin
    a:=0;
    locate:=nill;
    point_:=list;
    if point_<>nill then
         repeat
         if node[point_].data<>data_
           then
             point_:=node[point_].link
             else begin
             a:=a+1;
             if a=n then begin
             locate :=point_;
             exit;
               end;
         until point_=list;
  end;
```

Soru-3: Bir bağlı doğrusal dizide n kadar hücrenin kopyalanmasını sağlayınız.

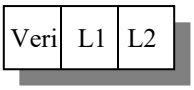
```
Function copy(list,n:integer):integer;
  Var
    Pointy,suret,addhead:integer;
  Begin
    a:=0;
    suret:=nill;
    if list<>nill then
      repeat
        a:=a+1;
        if a<=n then
          concetanete(suret,cons(node[list].data,nill));
          list:=node[list].link;
      until list=nill;
      copy:=suret;
  end;
```

Soru-4: Soru3'ü bir bağlı dairesel için tekrar çözünüz.

```
Function copy(list,node:integer):integer;
  Var
    Pointy,suret,a:integer;
  Begin
    a:=0;
    suret:=nill;
    pointy:=list;
    if pointy<>nill then
                repeat
                 a:=a+1;
               if a<=node then
  concatenate(suret,cons(node[pointy].data,node[pointy].link));
              pointy:=node[pointy].link;
            until pointy=list;
            copy:=suret;
  end;
```

2.3 İKİ BAĞLI DOĞRUSAL LİSTELER

Tek bağlı listelerden farklı olarak forwardlink(ileri yönde bağlantı), backwardlink(geri yönde bağlantı) vardır.

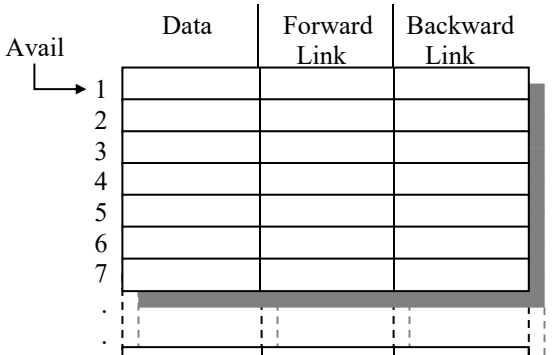


flink bir sonraki node'u gösterir (forward link).

blink bir önceki node'u gösterir (backward link).

Unit Node_uz

Burada aynı tek bağlı listelerde kullandığımız node_uz unitine benzer şekilde bir unit kullanıyoruz. Fakat burada bir tane sonraki node'u gösterecek, bir tane de önceki node'u gösterecek linkleri tanımlıyoruz. Bunların integer tipinde olması yeterlidir. Çünkü node uzayımız yine 10 adet node'dan oluşmaktadır.



```
Unit node_uz;

    Interface (**)

    Cons
      Memory=10;
      Nill=0;

    Var
      Avail: integer;
      Node: array[1...memory] of record
                                Data:real;
                                Blink:integer;
                                Flink:integer;
                                   End;
    Implementation (**)
End.
```

Procedure dumpmem

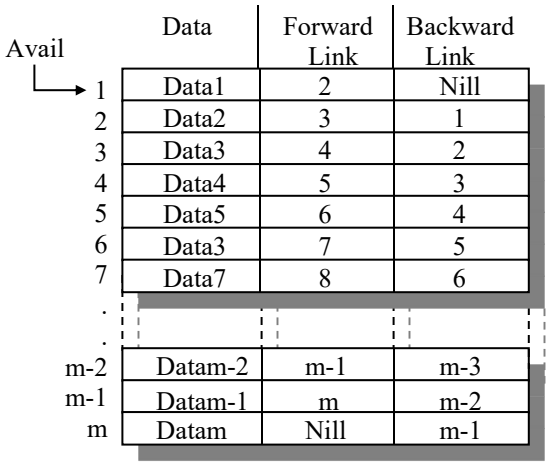
Node uzayımızın tüm data ve link'lerini ekrana basarak node uzayında ne olduğunu gösterir

```
Procedure dumpmem;

  Var
    i: integer;
  Begin
    For i:= 1 to memory do
    Writeln('node[',1:3,']=',node[i].data ' , '
node[i].flink', ' node[i].blink');

  End;
```

Procedure nodeinit

Bu procedure linklerin birbirine bağlanmasını Şekil 1.5'deki gibi bir node uzayının oluşmasını sağlar. Burada her flink bir sonraki node'u gösterecek şekilde tanımlanıyor (node[i].flink :=i+1). Ayrıca her blink bir önceki node'u göstereceğinden node[i].blink :=i-1 şeklinde bir tanımlama getiriliyor. İki bağlı doğrusal dizi kullanacağımızdan dolayı en son node'un linkini nill'e eşitledik (node.[memory].flink :=nill). Ayrıca node[avail].blink :=nill diyerek

```
Procedure nodeinit;

  Var
    i: integer;
  Begin
    Avail:= 1;
    For i:= 1 to memory-1 do
           Node[i]. flink:= i+1;
    Node[memory]. flink:=nill;
    Node[1].blink:=nill;
    For i:=memory down to 2 do
    Node[i].blink:=i-1;
  End;
```



Function newnode

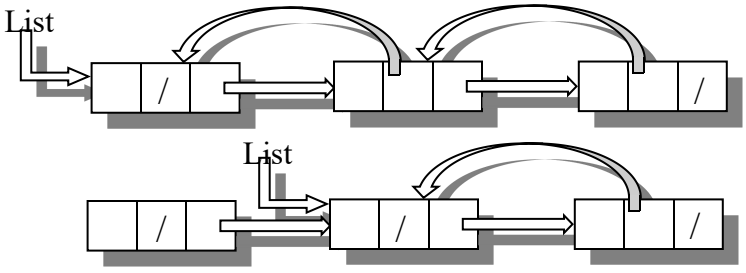
Newnode fonksiyonunda linklerle ilgili bir işlem yapmadığımızdan dolayı tek bağlı listelerdeki fonksiyondan farklı olarak bir şey yapmıyoruz.

```
Function newnode: integer;

  Var
    New: integer;
  Begin
    New:= cuthead(avail);
         If  new= Nill Then
              Begin
                    Writeln('nodeSpace Full...');
                    Halt;
              End;
            {Else}
         Newnode:= new;
  End;
```

Function cuthead

Listenin ilk node'unu koparıyor.

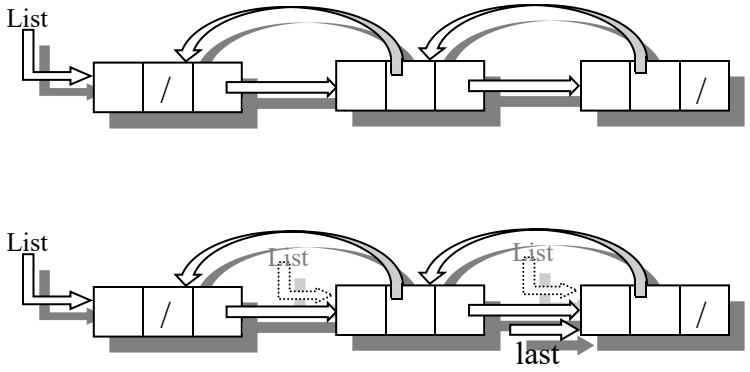
```
Function cuthead(var list: integer): integer;

  Begin
    Cuthead:= list;
    If   list <> nill   then
         If  node[list]. flink<>nill Then
           Begin
              Node[node[list].flink].blink:=nill;
           List:= node[list].link;
           End;
  End;
```



Function last

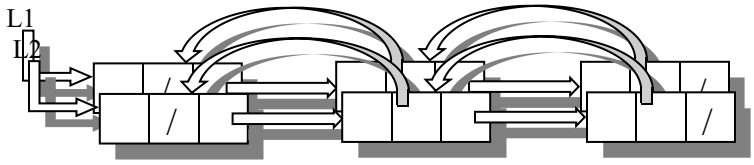
last fonksiyonu bize listemizdeki en son node'u veriyordu. O halde bu fonksiyonumuzda bir değişiklik yapmamız gerekecek. Link yerlerine flink yazarak bu değişikliği gerçekleştiriyoruz. En sonda da last :=list diyerek fonksiyonumuzun dönüşte bize list'i göndermesini sağlıyoruz.

```
Function last (list: integer): integer;
  Begin
    If  list<> nill Then
      If  node[list]. flink<>nill Then
           While node[list].flink<>nill do
              Begin
                 List:= node[list].flink;
      End;
    Last:=list;
  End;
```



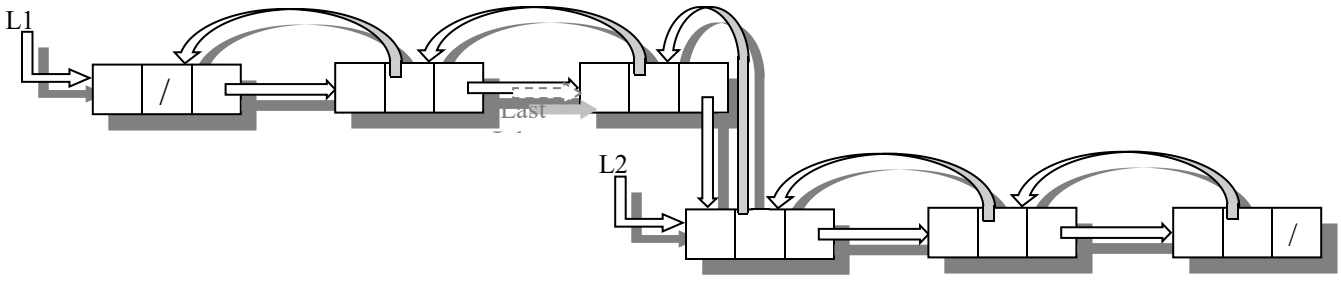
Procedure concatenate

Diyelim ki elimizde iki bağlı L1 ve L2 listeleri var. Bunları birleştirmek için kullandığımız fonksiyondur.



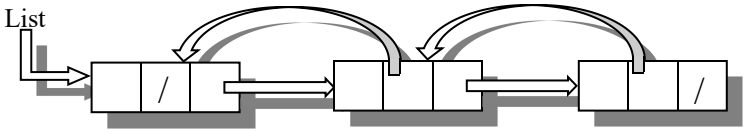
Örneğin L2'yi L1'in arkasına ekliyoruz, eğer L1 boşsa. L1 dolu ise, L1'in en son node'unu buluyoruz. Daha sonra L1'in last'ı L2 olsun diyoruz. Tek bağlı diziden farklı olarak node[L2].blink :=list(L1) diyerek

```
Procedure concatenate(var L1: integer; L2: integer);
  Begin
    If  L2<>nill Then
        If  L1=nill Then    L1:=L2;
    Else
        Begin
          Node[L2]. Blink:=last[L1];
          Node[last(L1)]. Flink:=L2;
        End;
  End;
```

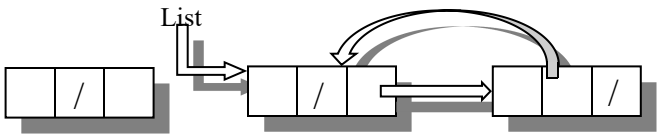


Procedure free

Free fonksiyonunu kullanmadığımız diziyi avail dizisine eklemek için kullanıyoruz. Bu fonksiyon tek bağlı dizi yapısındaki free procedure'ü ile aynı yapıdadır.



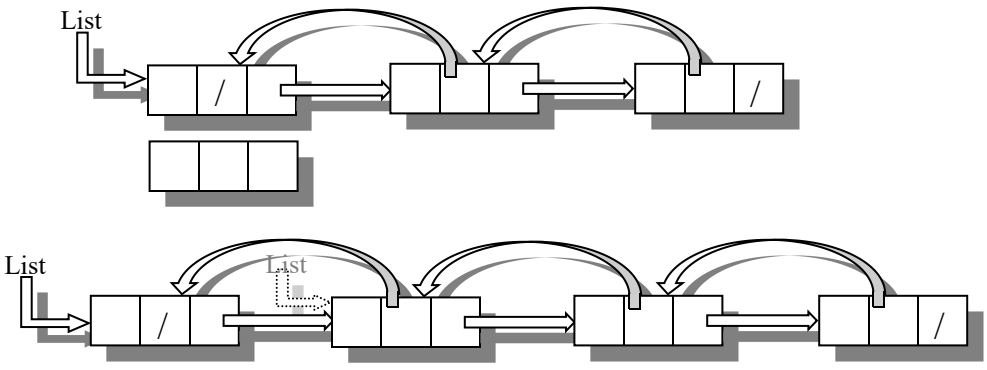

```
Procedure free(list: integer);
  Begin
    Concatenate(list,avail);
    Avail:= list;
  End;
```



Procedure addhead

İki bağlı doğrusal listelerde diziye istenen node'un eklenmesini sağlayan program.

```
Procedure addhead(node_: integer; var list: integer);

  Begin
    If  list=nill Then
         Begin
           Node[node_].flink:= nill;
           Node[node_].blink:=nill;
         End;
    Else
         Begin
           Node[list].blink:=node_;
           Node[node_].flink:=list;
           Node[node_].blink:=nill;
         End;
    List:=node_;
  End;
```



Function cons

Newnode ile yeni bir node alınıp cons'a gönderilen data ve link değerleri bu node'a aktarılıyor.

```
Function cons (data_:real; flink_.blink_:integer):integer;

  Var
    Cons_: integer;
  Begin
    Cons_:newnode;
    Cons: cons_;
    Node[cons_].data:=data_;
    Node[cons_].flink:=flink_;
    Node[cons_].blink:=blink_;
  End;
```

Function copy

İki bağlı doğrusal listelerde bir dizinin kopyasını elde etmeyi sağlayan program.

```
Function copy(list: integer): integer;
  Var
    Suret,point: integer;
  Begin
    Suret:= nill;
    Point:=list;
    If  list<>nill Then
         Repeat
           Concatenate(suret, cons(node[point].data, nill));
           point:= node[point].flink;
         Until point= nill;
    Copy:= suret;
  End;
```

Function locate

İki bağlı doğrusal listelerde aranan node'un yerini bildiren program.

```
Function locate(data_, list: integer): integer;
  Begin
    Locate:= nill;
    While list <> nill do
         If node[list].data <> data_ Then  list:=
node[list].flink;
         Else
           Begin
             Locate:= list;
             Exit;
           End;
  End;
```

Function member

İki bağlı doğrusal listelerde istenilen node'un var olup olmadığını bildiren program.

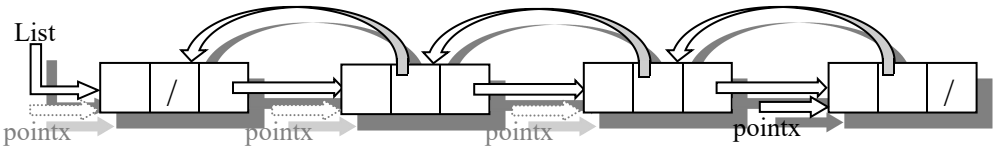
```
Function member(node_, list: integer): boolean;

Begin
     While (list <> nill) and  (list <> node_) do
```

```
        List:= node[list].flink;
        Member:= list=node_;
    End;
```

Function advance

İki bağlı doğrusal listelerde bir node ileri gitmeyi sağlayan program.

```
Function advance(var point: integer): boolean;

  Begin
    Advance:= false;
    If  list <> nill Then
        If  node[point].flink <> nill Then
          Begin
            Point:= node[point].flink;
            Advance:= true;
          End;
  End;
```

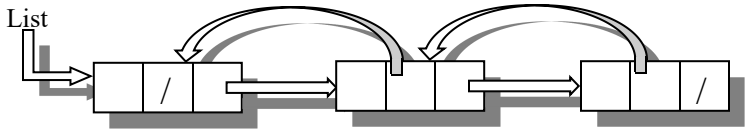


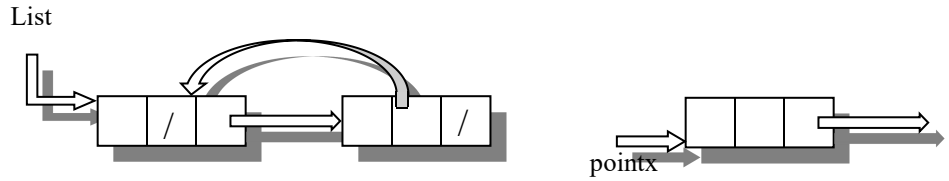
Function delete

İki bağlı doğrusal listelerde istenilen node'u silen program.

```
Function delete(node_: integer; var list: integer): boolean;

  Var
    Point: integer;
  Begin
    Delete:= false;
    If  list= nill Then exit;
      If  list= node_ Then
        Begin
          Addhead (cuthead(list), avail);
          Delete:= true;
```

```
        End;
      Else
        Begin
          Point:= list;
          While node[point] .flink <> nill do
            Begin
                  If node[point].flink= node_ then
                  Begin
            Addhead (cuthead (node[point].flink), avail);
                  Delete:= true;
                  Exit;
                  End;
                  Else point:= node[point] .flink;
            End;
        End;
  End ;
```





2.4 İKİ BAĞLI DAİRESEL LİSTELER

Unit node_uz

```
Unıt node_uz;

Interface (**)

Cons
       Memory=10;
       Nill=0;
```

```
Var
       Avail: integer;
         Node: array[1...memory] of record
              Data:real;
              Blink:integer;
              Flink:integer;
          End;
Implementation (**)
                    End.
```

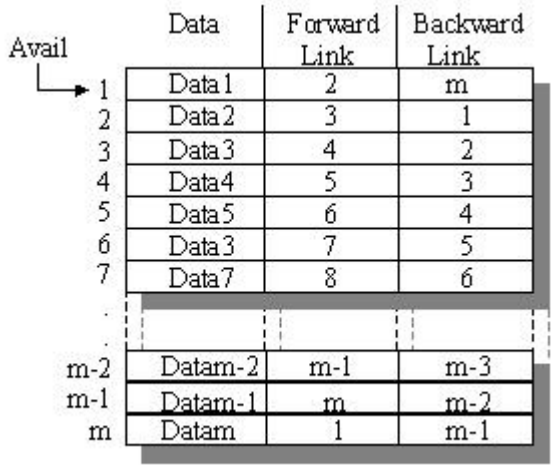
Procedure dumpmem

```
Procedure dumpmem;
  Var
    i: integer;
  Begin
    For i:= 1 to memory do
    Writeln('node[',1:3,']=',node[i].data ' , '
node[i].flink', ' node[i].blink');
  End;
```

Procedure nodeinit

İki bağlı dairesel listelerde node'ları birbirlerine bağlama.

```
Procedure nodeinit;

  Var
    i: integer;
  Begin
    Avail:= 1;
    For i:= 1 to memory-1 do
      Node[i]. flink:= i+1;
    Node[memory]. flink:=1;
    Node[1].blink:=memory;
    For i:=memory down to 2 do
      Node[i].blink:=i-1;
  End;
```



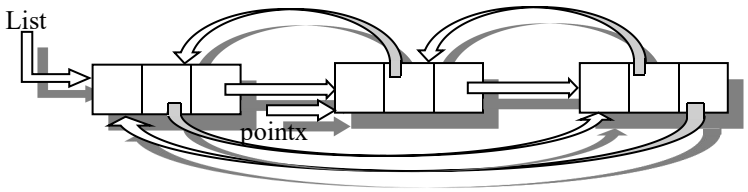
Function newnode

İki bağlı dairesel listelerde listeye yeni bir node eklemeye yarayan program.

```
Function newnode: integer;

  Var
    New: integer;
  Begin
    New:= cuthead(avail);
    If  new= Nill Then
      Begin
        Writeln('nodeSpace Full...');
        Halt;
      End;
  {Else}
    Newnode:= new;
  End;
```

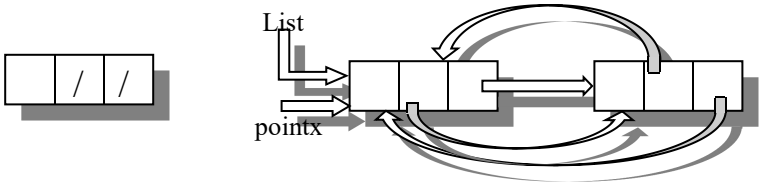
Function cuthead

İki bağlı dairesel listelerde dizinin ilk node'unu silen program.




```
Function cuthead(var list: integer): integer;

  Begin
```

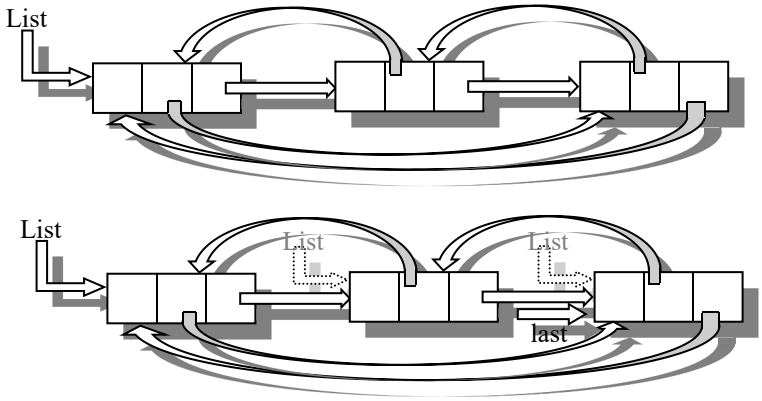
```
    Cuthead:= list;
    If   list <> nill   Then
      If node[list]. flink=list then list:=nill
        Else if list<> nill Then
        Begin
          Node[node[list]flink].blink:= node[list]. blink;
          Node[node[list]blink].flink:= node[lisy]. flink;
        List:=node[list]. flink;
        End;
  End;
```



Function last

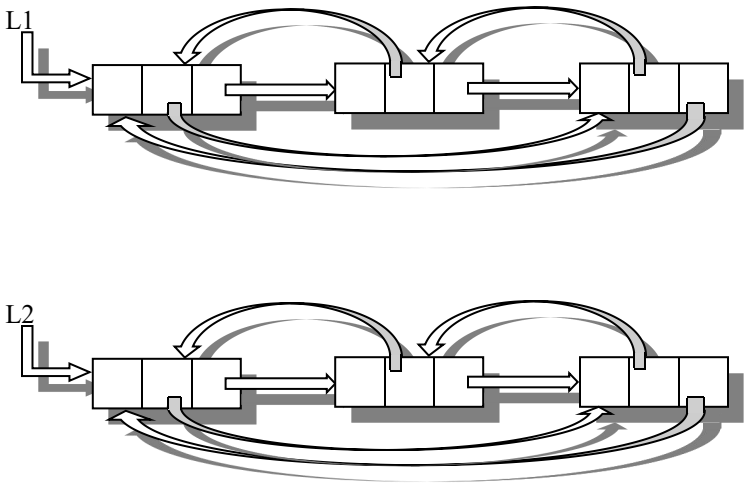
İki bağlı dairesel listelerde dizinin son node'unu bulan program.

```
Function last (list: integer): integer;

  Begin
       Last:=list;
       If  list<> nill Then
       If node[list]. flink<>list Then
       Last:=node[list]. blink;
  End;
```

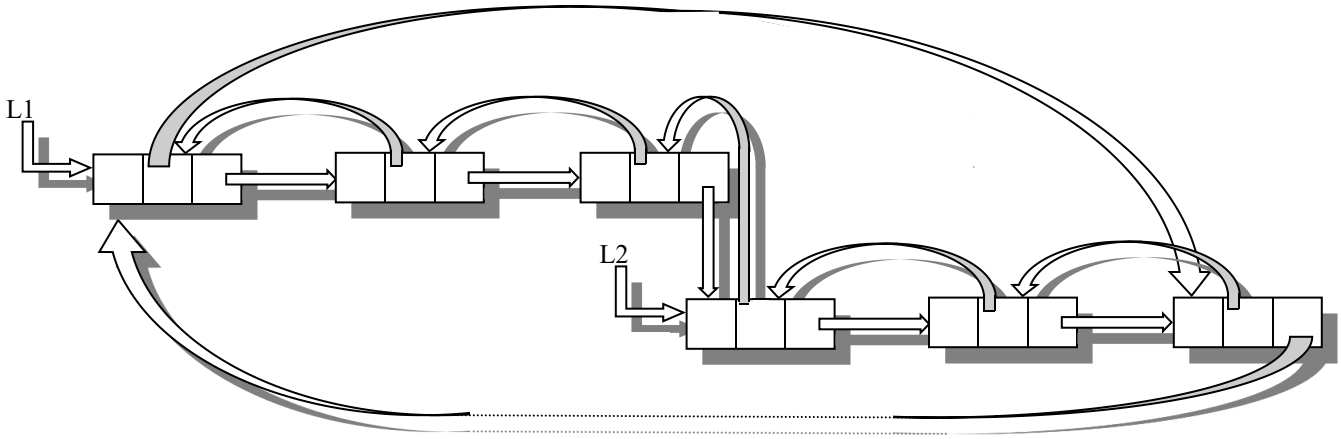


Procedure concatenate

İki bağlı dairesel listelerde iki diziyi birleştiren program.




```
Procedure concatenate (var L1:integer; L2:integer);

  Var
    Point:integer;
  Begin
    If  (L2<>nill) and (L1=nill)  Then L1:=L2;
      Else
    *     Begin
           Point:=last(L1);
           Node[last(L1)].flink:=L2;
           Node[last(L2)].flink:=L1;
           Node[L1].blink:=last(L2);
           Node[L2].blink:=point;
         End;
  End;


*'dan itibaren alternatif program.

    Begin
      Point:= Node[L1].blink;
      Node[node[L1].blink].flink:=L2;
      Node[node[L1].blink].flink:=L1;
      Node[L1].blink:= node[L2].blink;
      Node[L2].blink:=point;
    End;
  End;
```

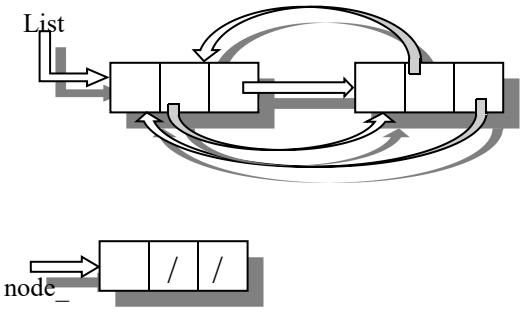


Procedure free

```
Procedure free(list: integer);
  Begin

    Concatenate(list,avail);
    Avail:= list;
  End;
```

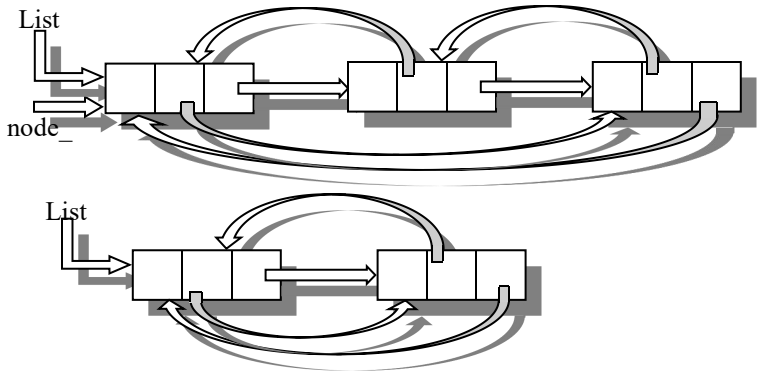
Procedure addhead

İki bağlı dairesel listelerde diziye yeni bir node ekleyen program.




```
Procedure addhead(node_: integer; var list: integer);

  Begin
```

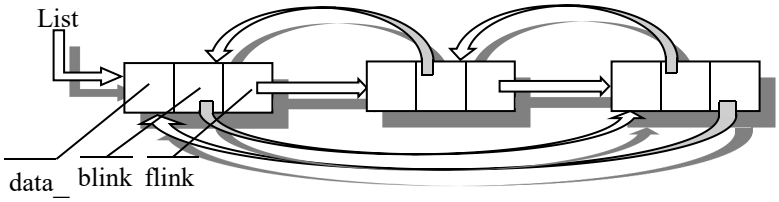
```
    If list:=nill Then
      Begin
        Node[node_].flink:= nill;
        Node[node_].blink:=nill;
      End;
            Else
      Begin
        Node[node[list].blink].flink:=node_;
        Node[node_].flink:=list;
        Node[node_].blink:=node[list].blink;
        Node[list].blink:=node_;
      End;
      List:=node_;
  End;
```



Function cons

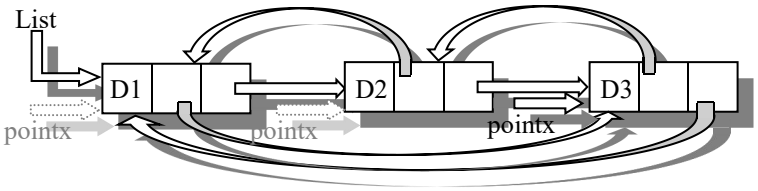
İki bağlı dairesel listelerde dizinin içeriğini tanımlayan program.


```
Function cons (data_:real; flink_.blink_:integer):integer;
  var
    Cons_: integer;
  Begin
    Cons_:=newnode;
    Cons:=cons_;
    Node[cons_].data:=data_;
    Node[cons_].flink:=flink_;
    Node[cons_].blink:=blink_;
  End;
```

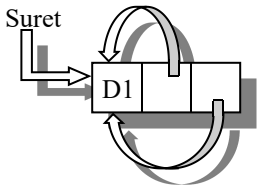


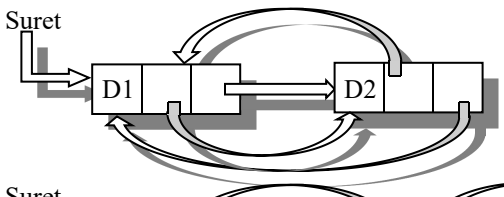
Function copy

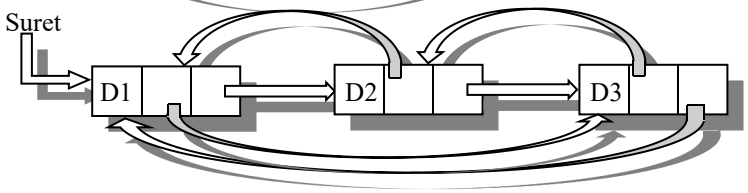
İki bağlı dairesel listelerde elimizdeki listenin kopyasını çıkarmamızı sağlaya program.




```
Function copy(list: integer): integer;

  Var
    Suret,point: integer;
  Begin
    Suret:= nill;
    Point:=list;
    If  list<>nill Then
      Repeat
        Concatenate(suret,cons(node [list].data, nill, nill));
        Point:=node[point].flink;
      Until point:=list;
    Copy :=suret;
  End;
```



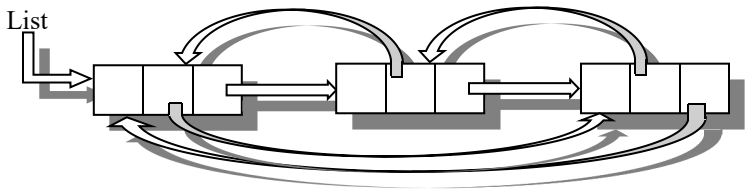


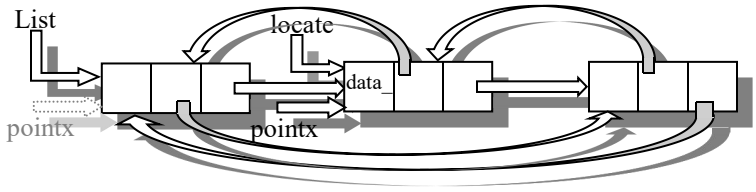


Function locate

İki bağlı dairesel listelerde bir node yerini bulmamızı sağlayan program.

```
Function locate (data_: real; list: integer): integer;

  Var
    Point:integer;

  Begin
    Locate:=nill; point:=list;
    If  list<>nill Then
      Repeat
        If  node[point].data<>data_  Then
          point:=node[point].flink;
        Else
          Begin
            Locate:=point;
            Exıt;
          End;
      Until point:=list;
  End;
```





Function member

```
Function member(node_, list: integer): boolean;
```

```
  Var
    Point:integer;
  Begin
    Point:=list;
    If  list<>nill  Then
      Repeat
        If  point<>node_  Then  point:=node[point].flink;
      Until (point=list) or (point=node_);
    Member:= point=node_;
  End;
```

Soru 5 : Locate fonksiyonunu öyle şekilde yazınız ki, fonkfiyona aktarılan data_ değerinin node_'de bulunması durumunda (node_ değişkeni de fonksiyona aktarılmaktadır.) mantıksal pozitif, aksi takdirde ise mantıksal negatif bir değer döndürsün?

Cevap:

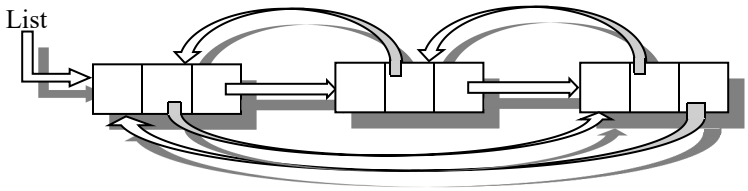
```
Function locate(data_:real;node_:integer;list:integer):boolean;
  Var
    Pointx, node:integer
  Begin
    Pointx:=list;
    Locate:=false;
    If list<>nill then
      Repeat
        If (node[pointx].data=data_) and (pointx=node_) then
          begin
            Locate:=true;
            Exit;
          End;
        Else pointx:=node[pointx].flink;
      Until pointx:=list;
  End;
```

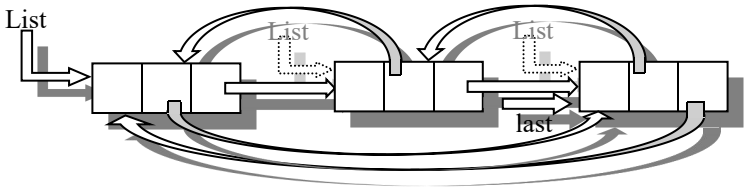
Function advance

İki bağlı dairesel listelerde bir sonraki node'a ilerlemeyi sağlayan program. Nill değilse ve kendisinden sonraki node kendisi değilse ilerler.

```
Function advance(var point: integer): boolean;
  Begin
```

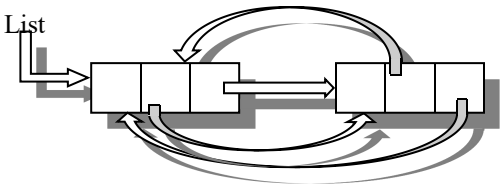
```
    Advance:= false;
    If  point <> nill Then
      If  node[point].flink <> point Then
           Begin
             Point:= node[point].flink;
             Advance:= true;
           End;
  End;
```

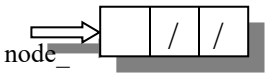




Function delete

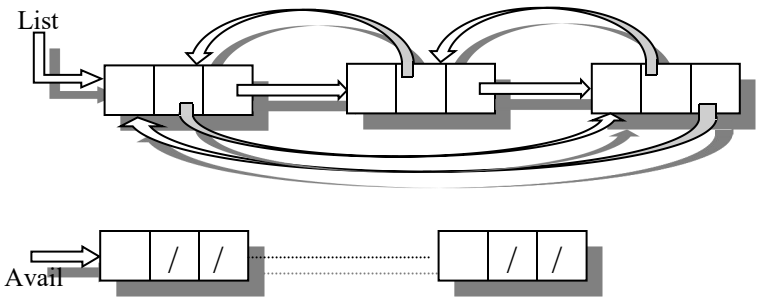
İki bağlı dairesel listelerde istenilen node'un silinmesini sağlayan program.






```
Function delete(node_: integer; var list: integer): boolean;
  Var
    Point: integer;
  Begin
    Delete:= false;
    If  list= nill Then exit;
    If  list= node_ Then
      Begin
        Addhead (cuthead(list), avail);
        Delete:= true;
      End
    Else
      Begin
```

```
        Point:= list;
        While node[point] .flink <> list do
             Begin
                  If node[point].flink= node_  Then
                        Begin
             Addhead (cuthead (node[point].flink), avail);
             Delete:= true;
             Exit;
                        End;
                  Else point:= node[point] .flink;
             End;
      End;
  End.
```



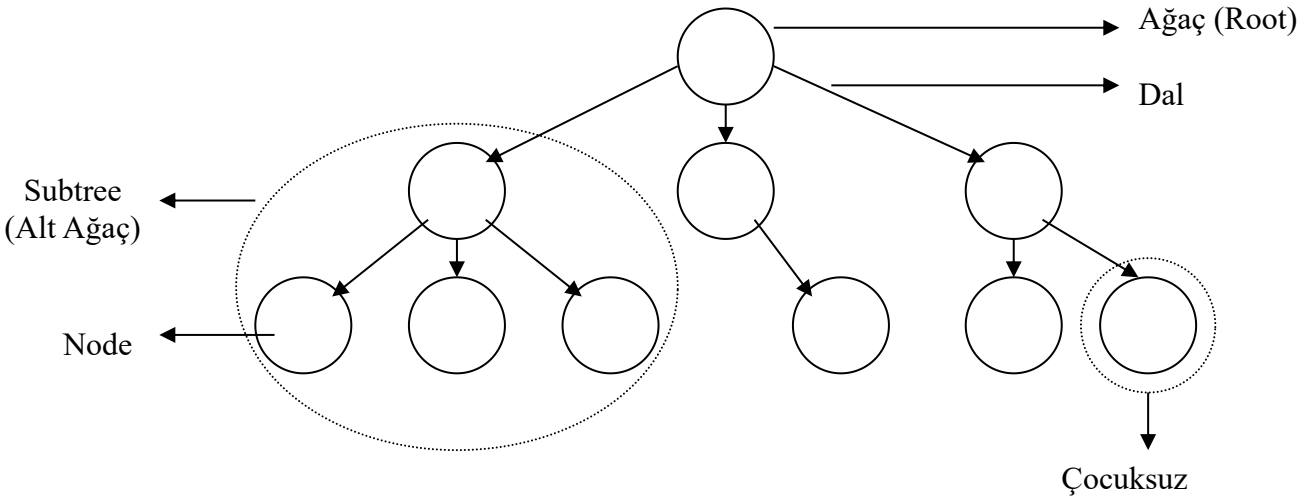
Soru-6: İki bağlı bir dizide node_'u n,n+1'inci node'lar arasına ekleyecek bir boolean fonksiyon yazınız?

Cevap:

```
Function insert(var list:integer; n, node_:integer):boolean;
  Var
    Pt:integer;
  Begin
    insert:=false; Pt:=list;
    İ:=1;
    İf  list<>nill then
         Repeat
           i :=i+1;
             İf i:=n then do begin
                   Node[node_].flink:=node;
                   [Pt].flink;
                   node[node_].blink:=Pt;
                   if node(Pt].flink<>nill then
                   node[node[pt].flink].blink:=node_;
                   node[Pt].flink:=node_;
                   insert:=true;
```

```
                    break;
                end;
            until (not (advance (Pt,List)));
end.
```

3. AĞAÇLAR

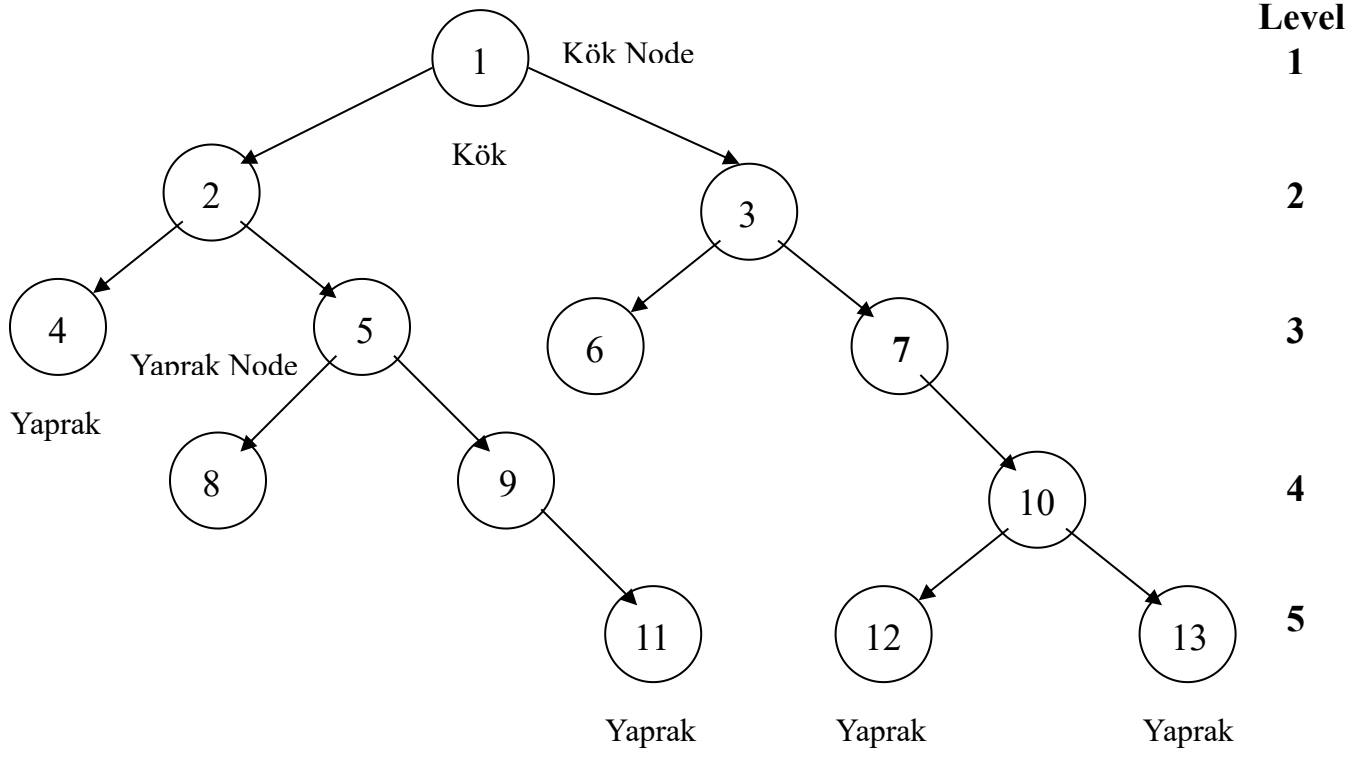


Bir Binary ağaçta, bir seviyedeki maximum node sayısı **2^{i-1}'**dir. (i>=1 olmalıdır).

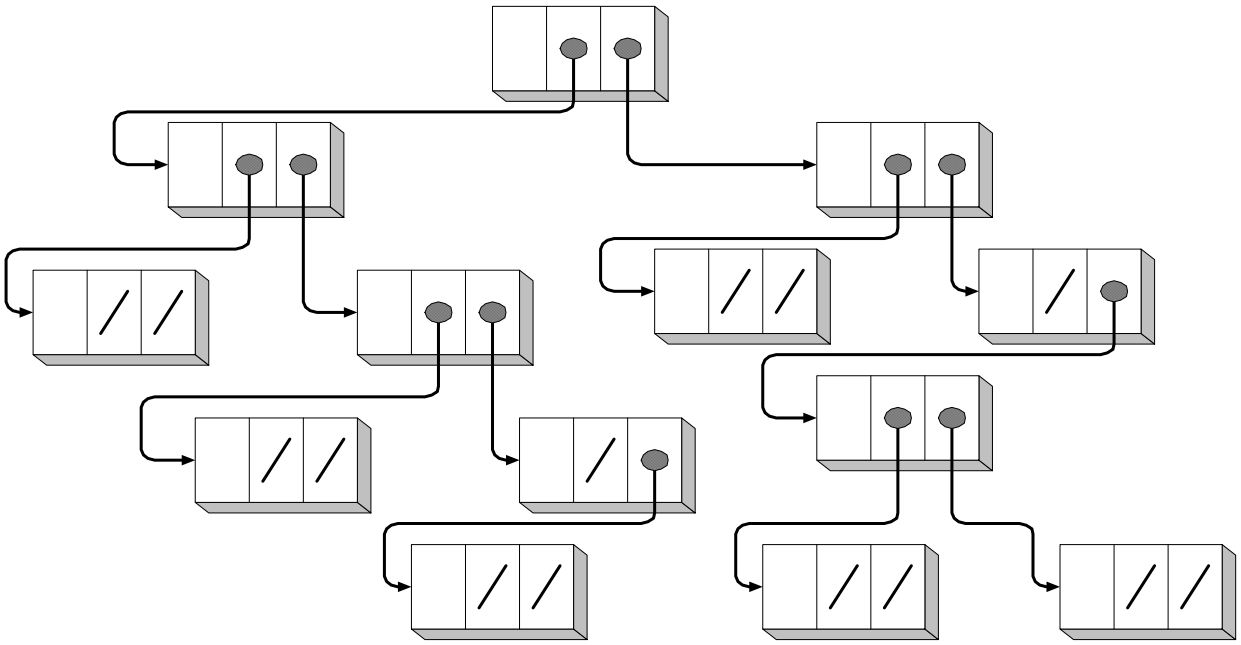
K derinliğine sahip bir binary ağaçta maximum node sayısı **2^{K}-1'**dir. (K>=1 olmalıdır).

Seviye ve derinlik kökle başlayacak şekilde aynı sayılmaktadır.

Ağaçta nodeların eksik kısımları nill olarak adlandırılır.

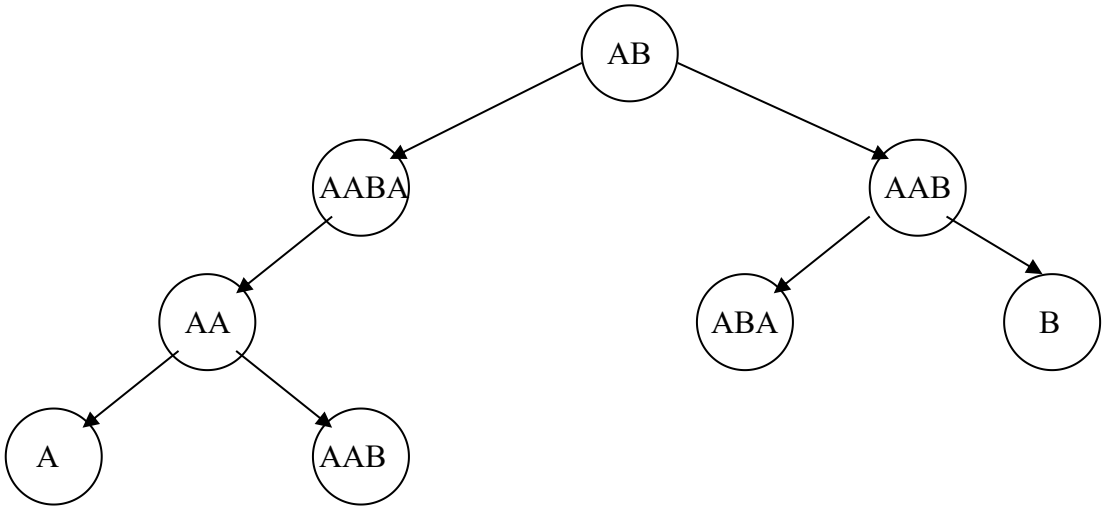


(Binary Tree)



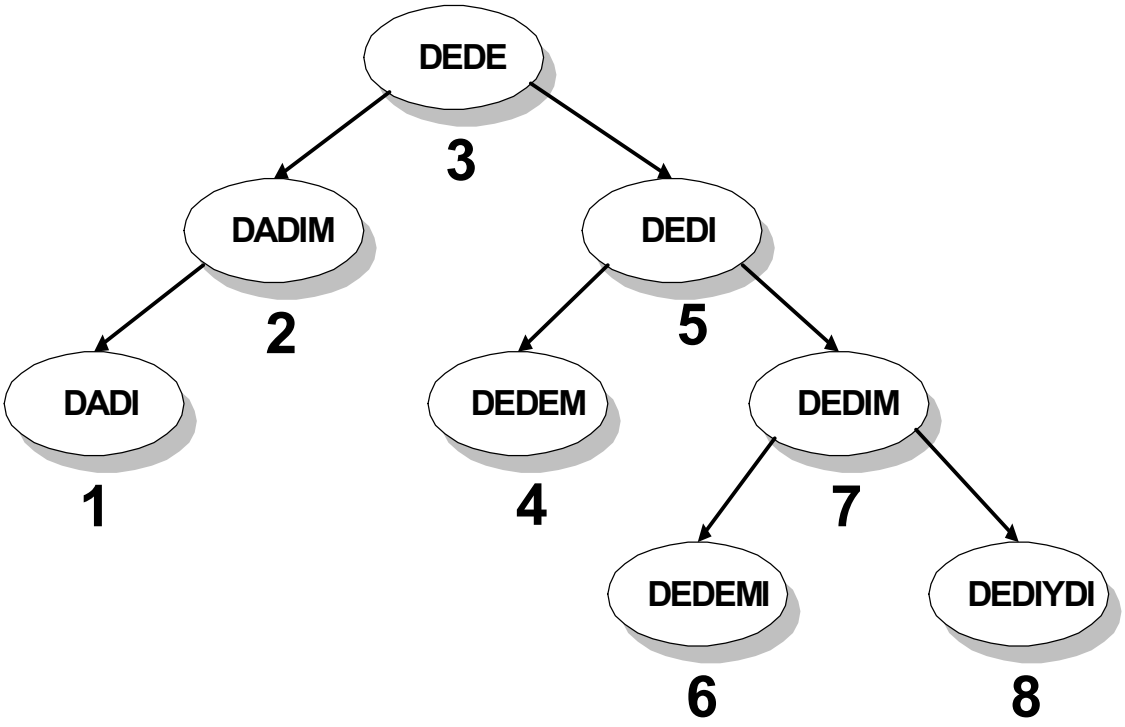
Soru: Aşağıdaki verilen geliş sıralarına göre nasıl bir sıralama ağacı oluşturacaklarını iki bağlı node'ların kullanıldığı bir ikilik ağaç çizerek (sola dayalı şekilde) gösteriniz.

AB,ABB,AABA,ABA,AA,B,A,AAB

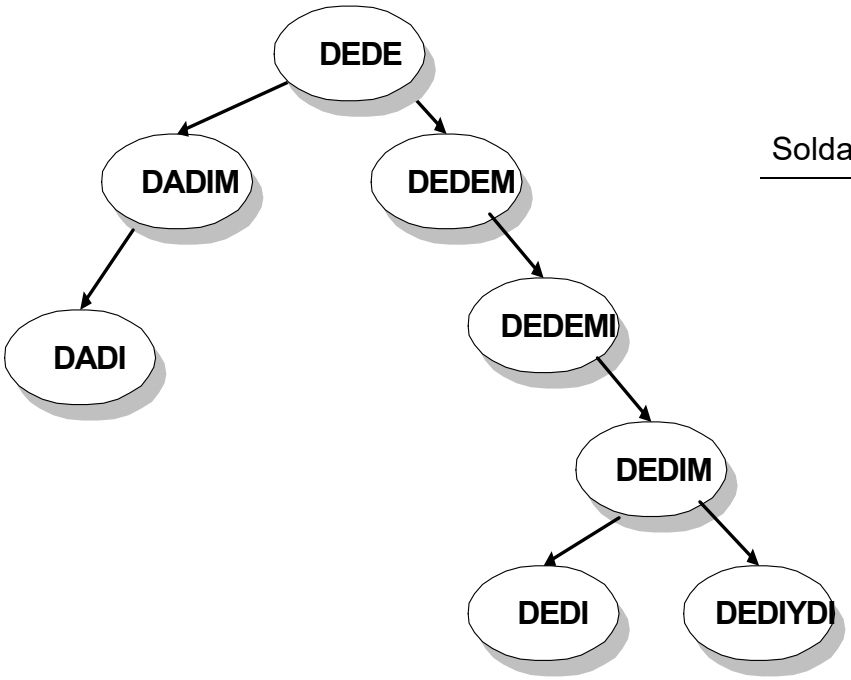


Soru: Aynı sıralama işlemini aşağıdaki dizilim için yapınız.

DEDE,DEDİ,DADIM,DEDİM,DADI,DEDEM,DEDİYDİ,DEDEMİ



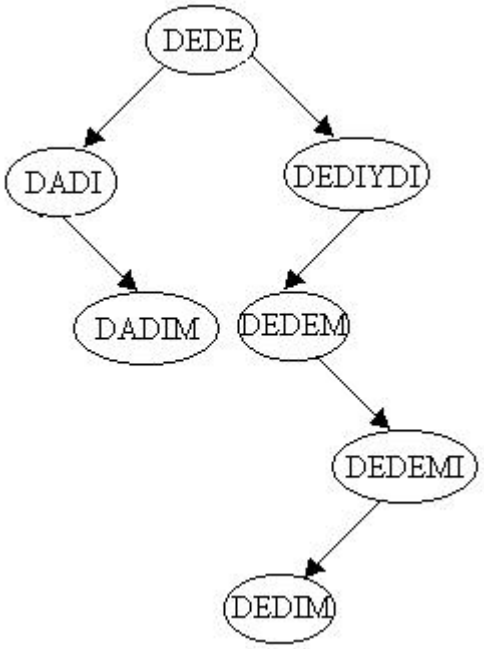
Soru: Sıralama işlemini aşağıdaki kelime dizilimine uygulayınız.



Soldan sağ tarafa	Sağdan sol tarafa
DADI DADIM DEDE DEDEM DEDEMI DEDI DEDIM DEDIYDI	DEDIYDI DEDIM DEDI DEDEMI DEDEM DEDE DADIM DADI

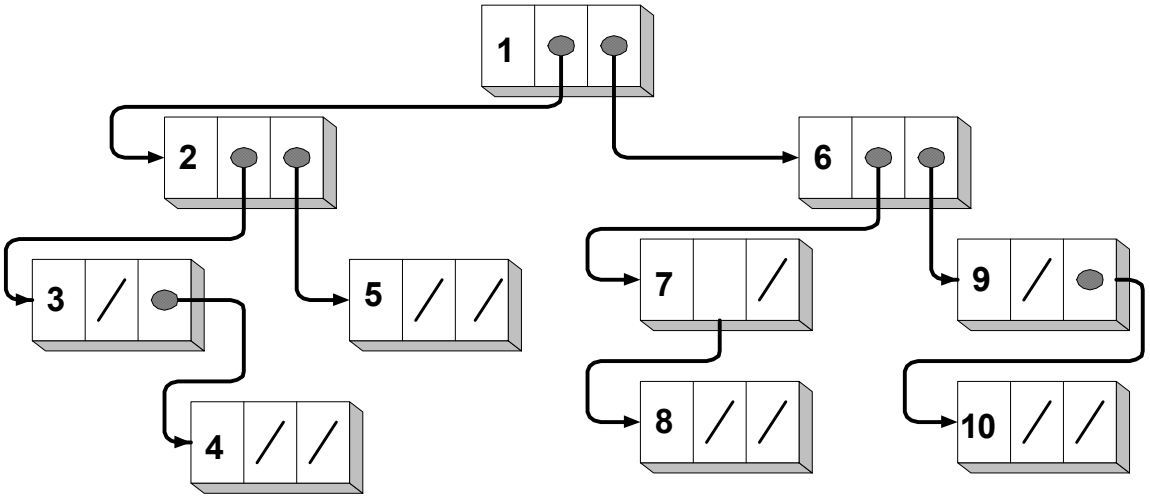
Soru: Aşağıdaki dizilimin ağaç yapısını çiziniz.

DEDE,DADI,DEDIYDI,DEDIM,DEDEM,DEDEMI,DEDIM



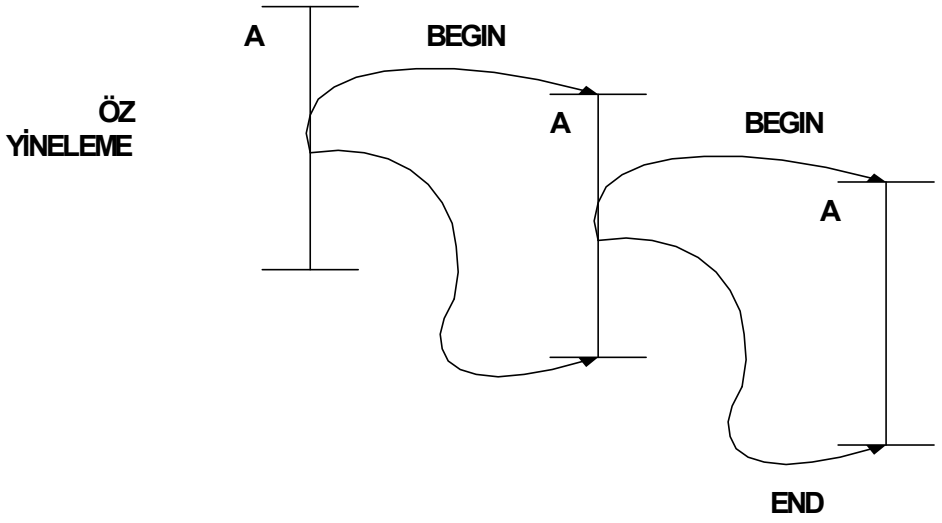
Soldan sağ tarafa	Sağdan sol tarafa
DADI	DEDIYDI
DADIM	DEDIM
DEDE	DEDEMI
DEDEM	DEDEM
DEDEMI	DEDE
DEDIM	DADIM
DEDIYDI	DEDI

***Printree** Procedure'ü*



```
Procedure Printree (t:integer);
Begin
     İf T<>nill then
Begin
     İf node[T].llink<>nill Printree(node[T].llink);
     Writeln (node[T].data);
     İf node [T].Rlink<>nill Printtree(node[T].Rlink);
End;
End;
```

RECURSION (Özyineleme): Herhangi bir program kendi kendini çağırıyorsa bu alt yordama "Recursion Subrotines" denir.



Soru: Verilen iki bağlı D1 doğrusal dizisinde, verilen bir X değerinden iki önceki veriyi yazan bir program parçası yazınız?

Burada ilk olarak X değeri aranmalıdır. Bulunduktan sonra 2 node geriye gidilip Overclock node'un verisi bulunur,Daha sonra X değerini taşıyan node'un 2 önceki verisine bakılır,X değeri hiç olmayabilir. Böyle bir durumda aranan data yoktur diye programdan mesaj verilecektir. Eğer X, 1 ve 2. node'da olursa o zaman 2 öncesinde bu node yoktur denilecektir.

CEVAP:

```
Function find_data(Data: real; list: integer): integer;
  Var
     Pointer: integer;
       Begin
         Pointer:nill;
         İf list<>nill Then
            If node [list].flink <> nill
   Then begin
         End;
       İf (node[list].data<>data_) then Pointer=nill;
       End;
     Find_data:=pointer;
  End;
```

Bir ağaçtaki sıralama şekilleri:

İnorder: Left tree'nin en sol ucundan başlayarak sağa tarama yapılır.

Preorder:Önce ana node sonra yan node'lar okunur.

Postorder: Sondan başa doğru sol öncelikli olarak tarama yapılır.

Ödev:

IN ORDER

```
Procedure Printtree(t:integer);
Begin
      If  t<>nill  Then
          Begin
            If  node[t].link <> printtree(node[t].link);
            Writeln (node[t].data);
            If  node[t].rlink <> nill printtree(node[t].rlink);
          End;
End;
```

PRE ORDER

```
Procedure Printtree(t:integer);
  Begin
    If  t<>nill  Then
      Begin
        Writeln (node[t].data);
```

```
        If  node[t].link <> printtree(node[t].link);
        If  node[t].rlink <> nill printtree(node[t].rlink);
      End;
  End;
```

POST ORDER

```
Procedure Printtree(t:integer);
Begin
If  t<>nill  Then
Begin
If  node[t].link <> printtree(node[t].link);
If  node[t].rlink <> nill printtree(node[t].rlink);
Writeln (node[t].data);
End;
End;
```

Bir Ağacın derinliğinin bulunmasını sağlayan Program:

```
Int Count (Mytree  *T)
{
  Static  int depth  =1 ; //Bu procedure her çağrıldığında bir
  Static  int  count =1 ; //önceki değeri kaybetmemesi için
   If (T.Llink) // static komutu kullanılır.
     {
      Counter++;
      Count (T.Llink);
     }
      if (T.Rlink)
       {
        Counter++;
        Count (T.Rlink);
       }
   If  ( counter > depth )
   Depth = counter;
  Counter - -;
 Return;
}
```

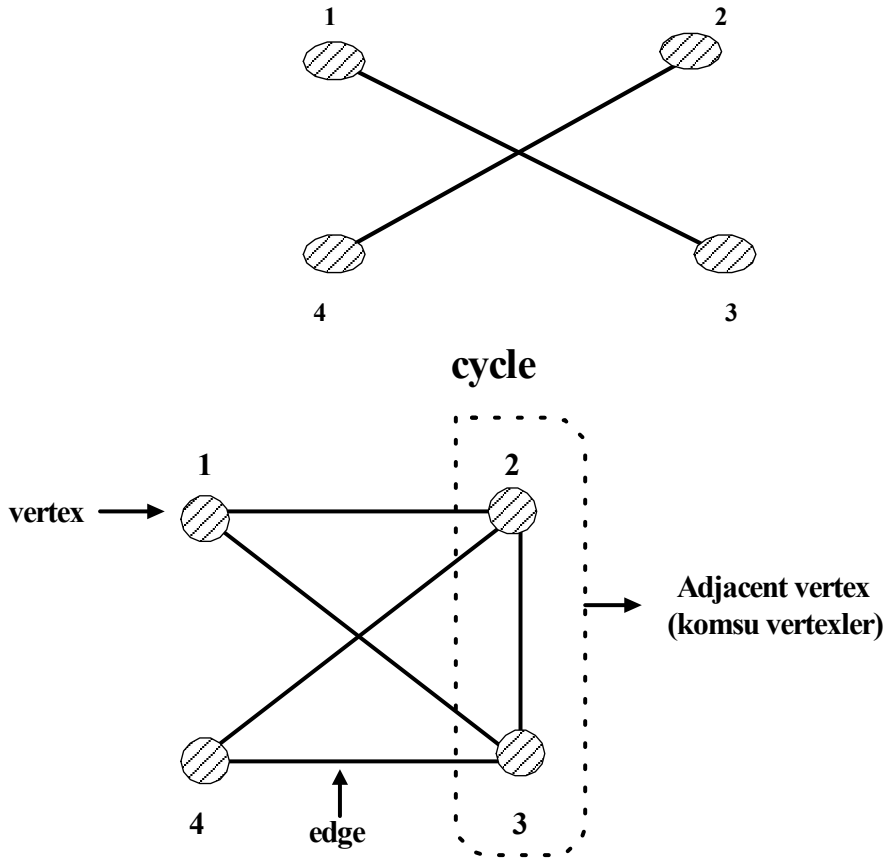
Static olarak tanımlana değerler; Program çağırılıp bittikten sonra tekrar çağırıldıklarında önceki tanımlamalarını korurlar. *T pointer'ı ağacın Root'u dur.

Sonra counter eksiltilerek geriye dönülüyor, önceki node'ların left ve right link,'i kontrol ediliyor.

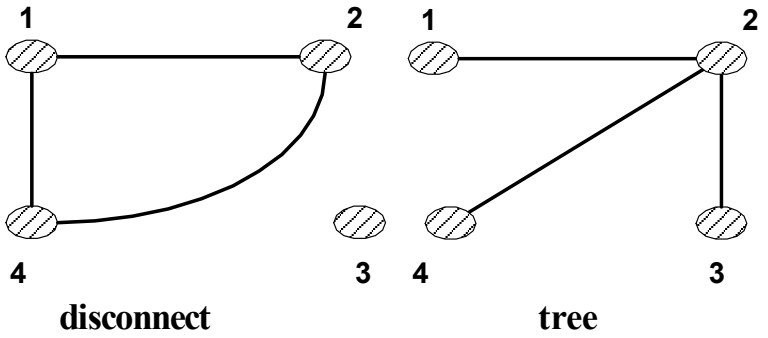
Sağ ve Sol Ağacın En derin yerini bulan program:

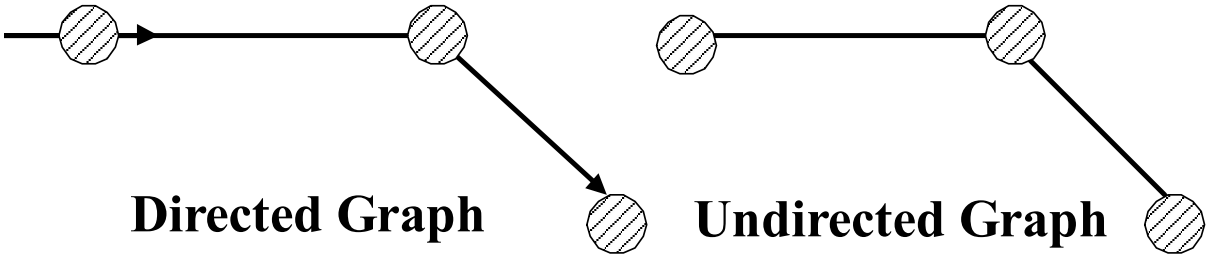
```
Int double count (mytree  *T)
 {
  static  int leftdepth, rightdepth ;
  int   counter =1 ;
  If   (T.Llink)
   {
    leftdepth =count (T.Llink);
    Leftdepth ++;
   }
  If (T.Rlink)
   {
    Rightdepth = count (T.Rlink) ;
    Rightdepth ++;
   }
 }
```

4. GRAPHS [ÇİZGE KURAMI]

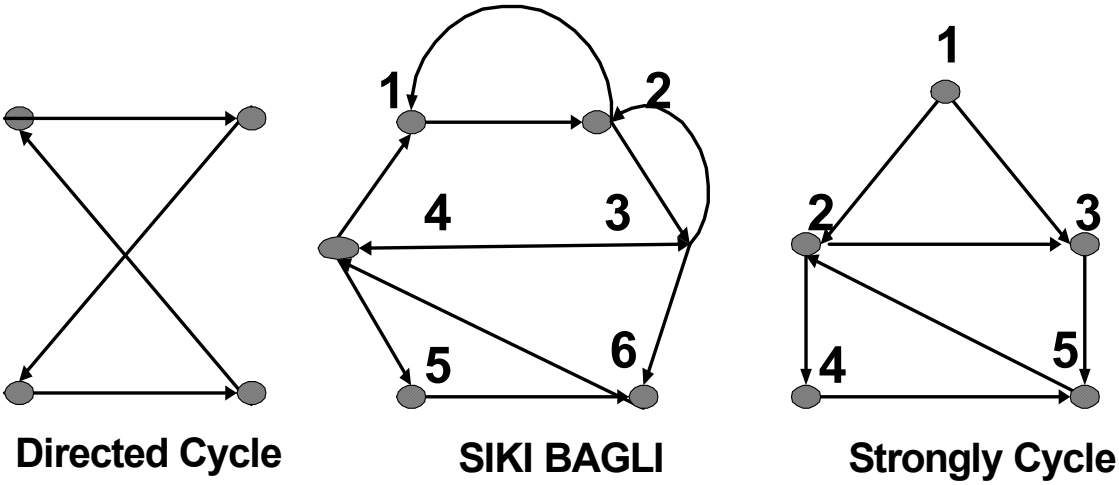


Bir "G" grafı bir "V" grubunun oluşturduğu Vertex'ler grubu ve bunlar arasında bağlantıyı sağlayan bir "E" kenarları grubundan oluşur. Buradaki kenarlar "G" nin kenarları diye adlandırılır.



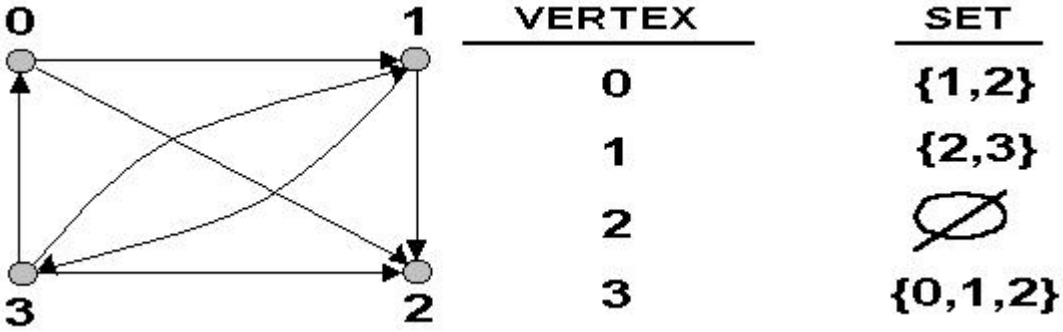


Directed Grap tek yönde haraketli olduğu halde Undirected graph her iki yönde de hareket eder, Path için weakly connected "gevşek bağlı" denir. Herhangi bir köşeden istediğimize gidiyorsak, bu strongly connected "sıkı bağlı"dır.

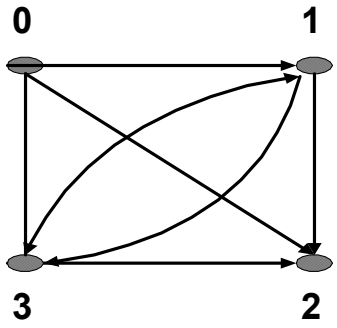


ADJACENCY SETS

(KOMŞULUK TABLOLARI)

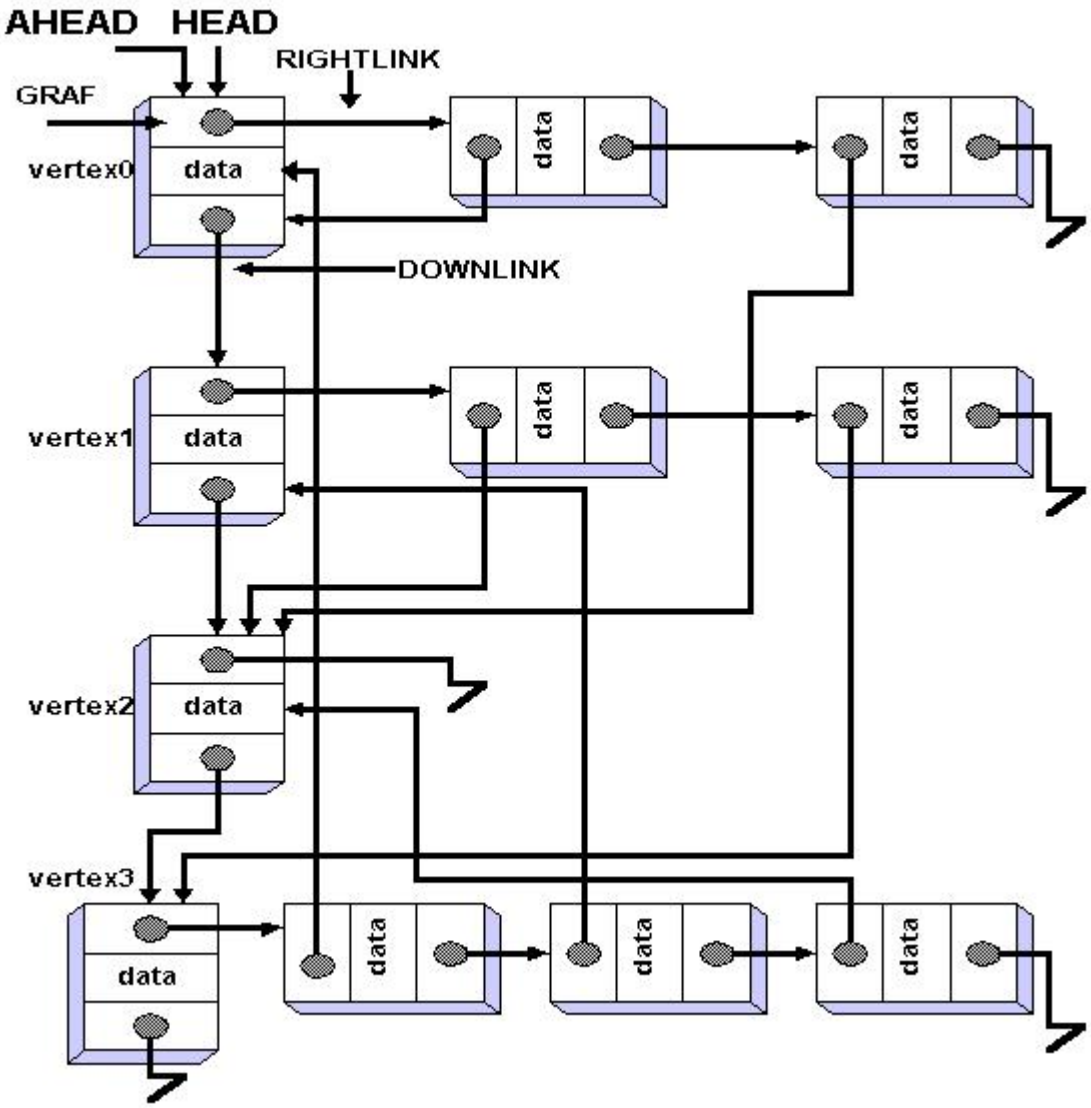


ADJOCENCY TABLE



LINKED LIST REPRESENTATION OF GRAPHS

(Grafların bağlı listeler ile temsil edilmesi)

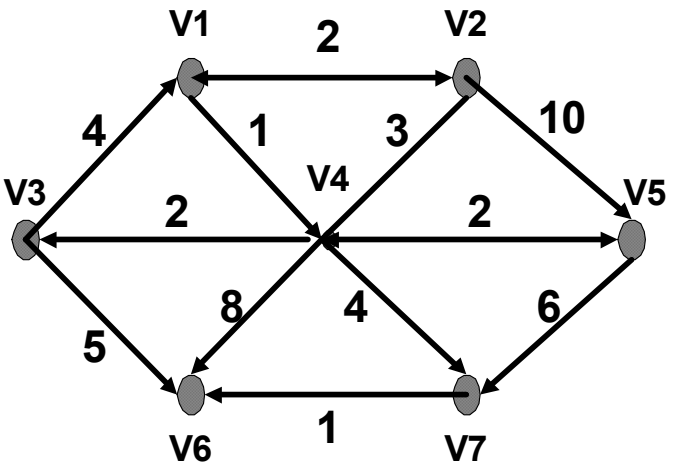


Soru: Diğer sayfa da tanımlanan yapıyı kullanarak n tane node'u bulunan bir graph'taki toplam adjocent node'ların sayısını döndüren programı yazınız.

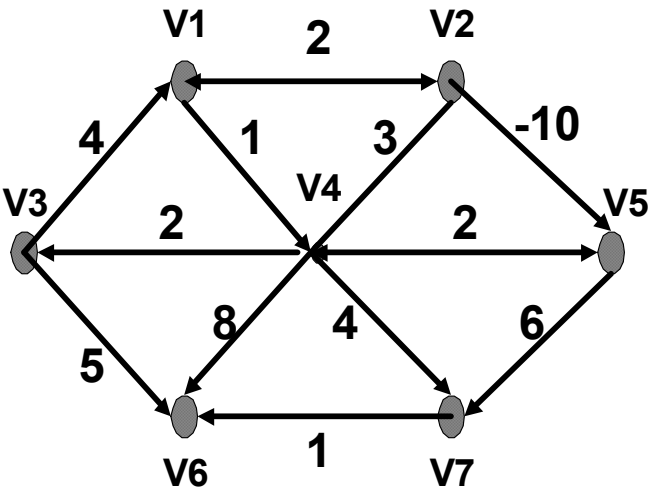
Not: Adjocent node'lar için vertex node'larında kullanılan aynı yapı ön görülmüş ve sonuncu node'un orlink'i nill olarak tanımlanmıştır.

Cevap:

```
Struct tree
  {
    int data;
    struct tree  *dlink;
    struct  tree * rlink;
  }
int func (struct tree  *lP)
  {
    int  count=0;
    struct tree  *head = *ahead= lp;
    while(head!=nill)
     {
       while  (ahead          rlink !nill)
        {
          count++;
          ahead=ahead           rlink;
        }
       head = head          dlink;
       ahead =head;
     }
return count;
  }
```



A directed weighted graph
Serbest path algoritması:



A directed weighted graph (with negative weight) V1-V4= cost cycle 1 denir....

En kısa yol problemi: Giriş olarak ağırlıklı bir G(V,E) graf verildiğinde belli bir S vertexinde diğer tüm vertexlere en kısa ağırlıklı yolun bulunması işlemidir.

Bakır plakalardaki iletken yolların belirlenmesinde, kısayol algoritması kullanılır.

Soru : N tane node bulunan bir graftaki toplam adjecent nodeların sayısını döndüren precedure'u yazınız?

Çözüm:

```
program node_sayisi_bul
uses crt;
const
  memory = 11;      nill = 0;
var
head, ahead, count : integer;
node : array [1 .. memory] of record
       data : real;
       rlink : integer;
       dlink : integer;
       end;
         procedure agac;
         begin
  node[1].rlink : = 2;          node[1].dlink : = 4;
  node[2].rlink : = 3;          node[2].dlink : = 4;
  node[3].rlink : = nill;       node[3].dlink : = 7;
  node[4].rlink : = 5;          node[4].dlink : = 7;
```

```
  node[5].rlink : = 6;          node[5].dlink : = 7;
  node[6].rlink : = nill;       node[6].dlink : = 8;
  node[7].rlink : = nill;       node[7].dlink : = 8;
  node[8].rlink : = 9;          node[8].dlink : = nill;
  node[9].rlink : = 10;         node[9].dlink : = 1;
  node[10].rlink : = 11;  node[10].dlink : = 4;
  node[11].rlink : = nill;      node[11].dlink : = 7;

  procedure node_sayisi ( N : integer );
  begin
   count :=0;
   head:=N;
   ahead:=N;
    While ( head <> nill )
       begin
         While( node[ahead].rlink <> nill )
           begin
             Count := count+1;
             Ahead := node[ahead].rlink;
           end;
         Head := node[head].dlink;   Ahead := head;
       end;
    Writeln ( 'Birbirine komşu olan nodeların sayısı=',count );
  end;
begin
  clrscr;
  agac;
  node_sayisi ( 1 );
  readln();
end;
```