Veri Yapıları

Dr. Günay TEMÜR

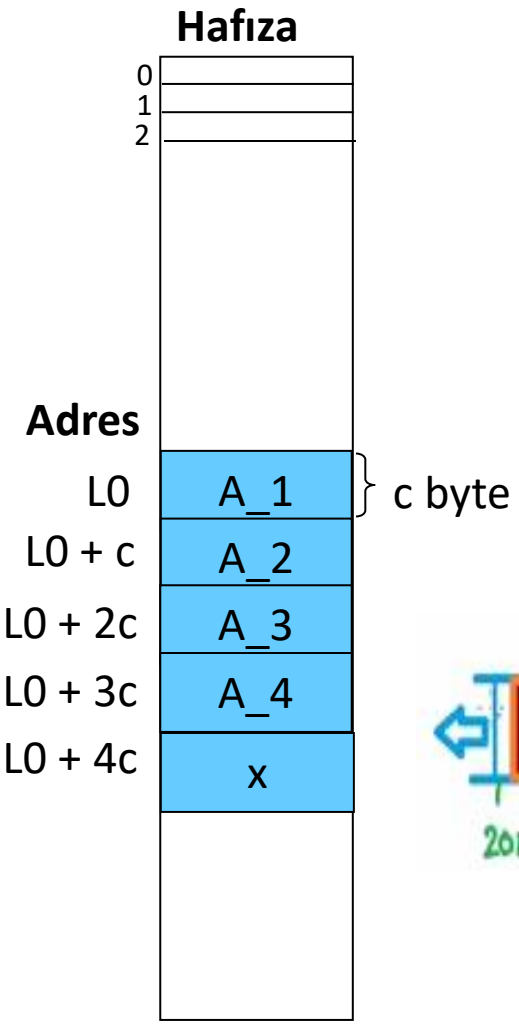
Düzce Üniversitesi

Neden Listeye İhtiyaç Var?

- **Dizi**, bellek yöneticisi tarafından atanan ve uzunluğu **baştan belli** **ardışık** bir hafızada saklanır.
- `int[] A = {6, 5, 4, 2}` şeklinde 4 elemanlı **int** türünde bir *dizi olduğunu varsayalım*.
- Hafızaya sırayla
  - `int x = 8;` sayısını ve
  - daha sonra A dizisini

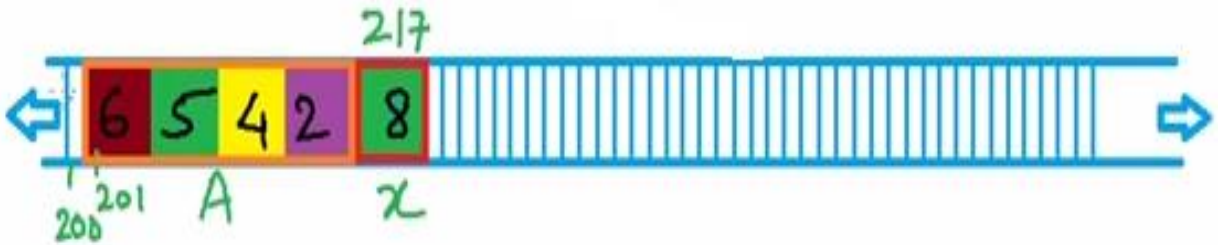
sakladığımızı düşünelim.

Neden Listeye İhtiyaç Var?




```
int x = 8;
int[] A = {6, 5, 4, 2};
```

Nasıl bir bellek organizasyonu bekliyoruz?



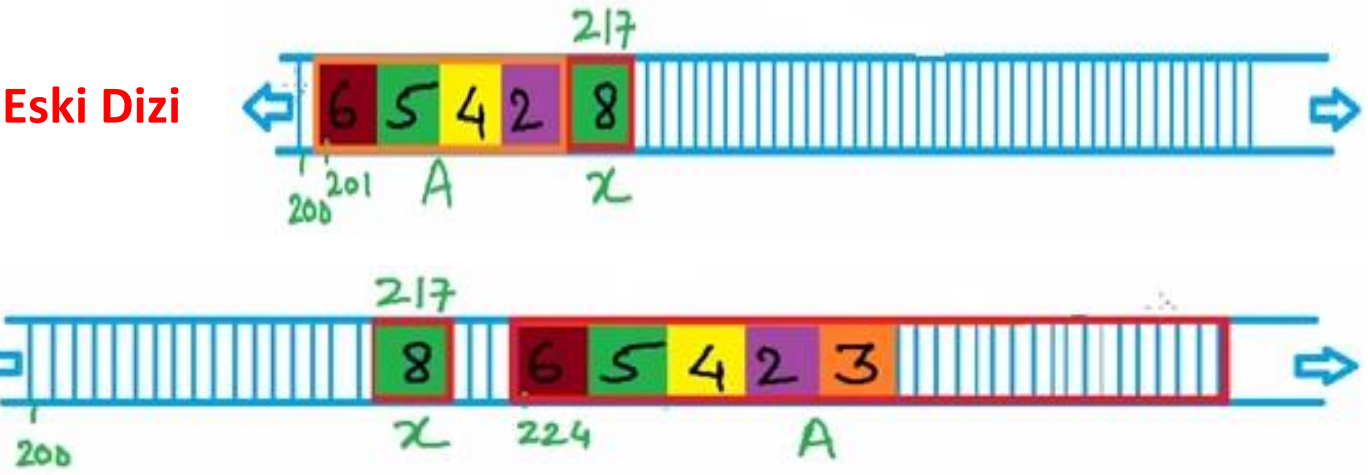
Neden Listeye İhtiyaç Var?

- Dizi için ayrılan bölümün sürekli adreslerden oluştuğuna dikkat edelim.
- Bu diziye **yeni bir eleman** (örneğin 3) **eklemek** istediğimizde
  - 2'den sonraki alan, yani 217 nolu hafıza dolu olduğu için hafızayı ***genişletme seçeneğimiz yoktur.***

- **Çözüm olarak ne yapabiliriz?**

Neden Listeye İhtiyaç Var?

- Çözüm olarak birçok **hafıza değişimi ve işlemi gerektiren** *aşağıdaki adımlar uygulanır*:
  - *Daha büyük bir dizi* için yer alanı oluşturulur.
  - Eski dizideki elemanlar yeni diziye **kopyalanır** ve taşıma (move) işlemi gerçekleştirilir.
  - **Eski dizi** hafızadan **silinir**.



Neden Listeye İhtiyaç Var?

- Bu problemi **çözmek** için ne yapabiliriz?

ÇÖZÜM 1: Büyük Hafıza Alanı

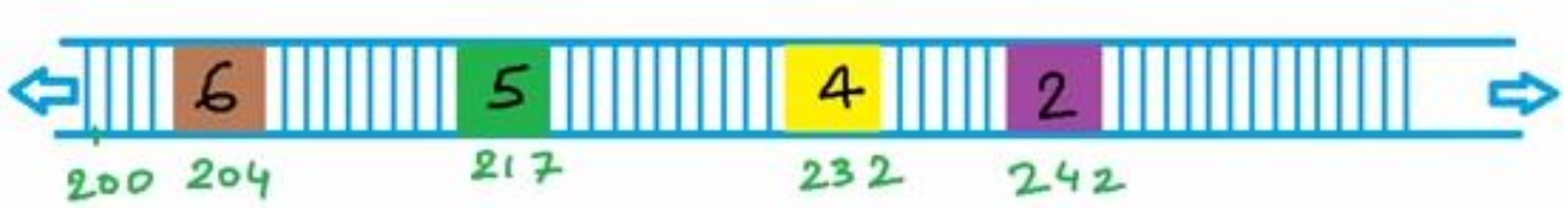
- Büyük bir hafıza alanını **dizi için ayırabiliriz**.
- Ancak bu durumda da hafızada saklanan eleman sayısı azken **hafıza boşa işgal edilmiş** olur.

ÇÖZÜM 2: Bağlı Liste

- Hafızayı ihtiyaç halinde büyütebilmektir.
- **Önceden alan ayırmayarak** hafızayı *verimli kullanmak adına* elemanları hafızaya tek tek yerleştirmektir.

Bağlı Liste – Çözüm

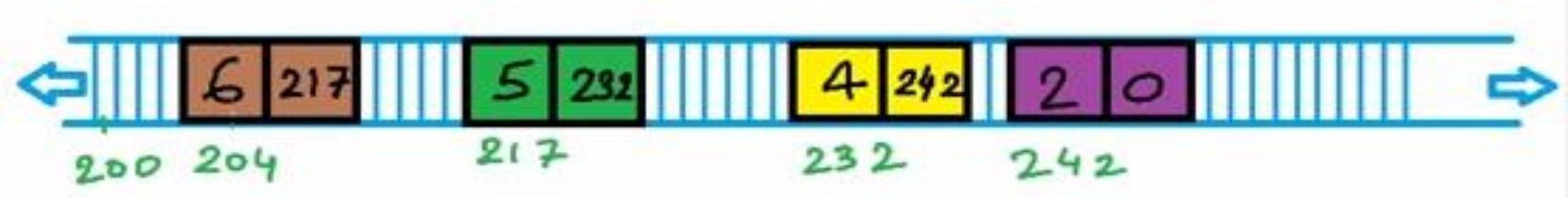
- `int[] A = {6, 5, 4, 2}` dizisini aşağıdaki gibi tutabildiğimizi varsayalım.



- Elemanlar arasında bir **bağlantı** sağlanması **zorunludur**. **Neden?**

Bağlı Liste – Çözüm (devam…)

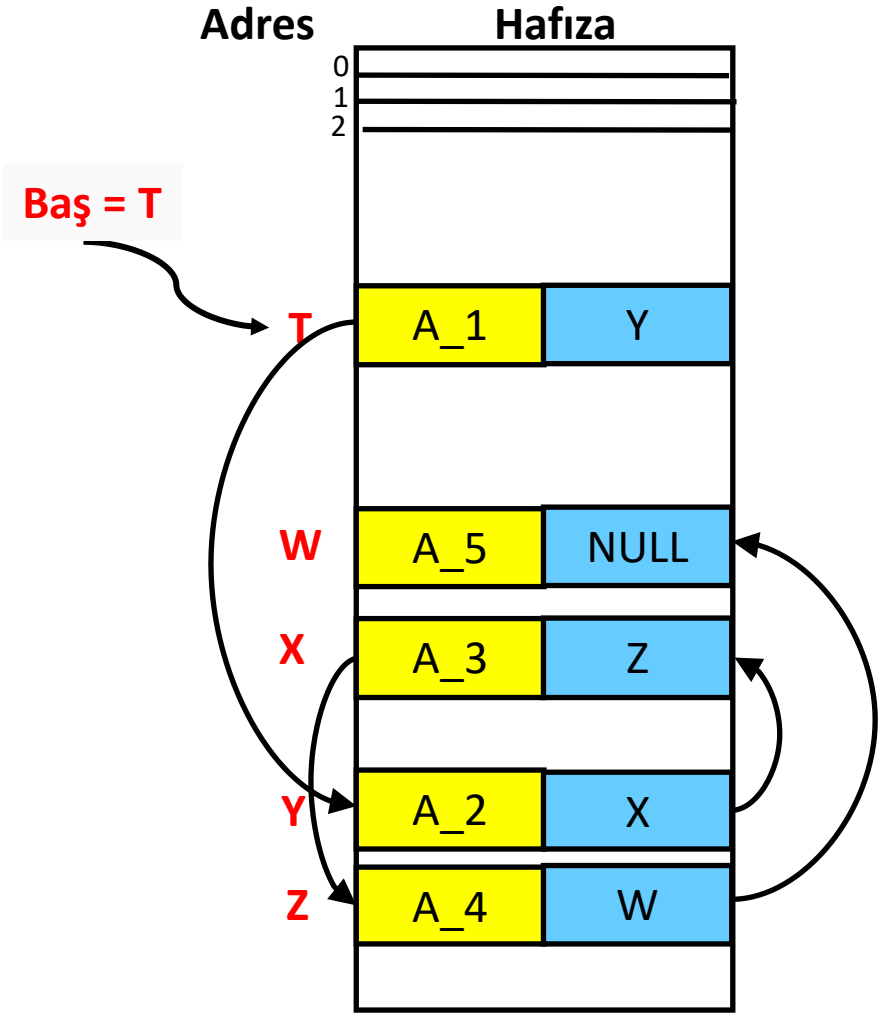
- Hafıza **sürekli olmadığı** için dizi elemanlarının birbirleriyle **bağı yoktur**. Bu nedenle her bir eleman ***sonrakinin adresini gösteren*** bir referansla bağlanmalıdır.



- Her eleman sonraki elemanın adresini tutarken **son elemanın adresi NULL** olur.
- Bu şekilde elde edilen veri yapısı, **Bağlı Liste (Linked List)** olarak adlandırılır.

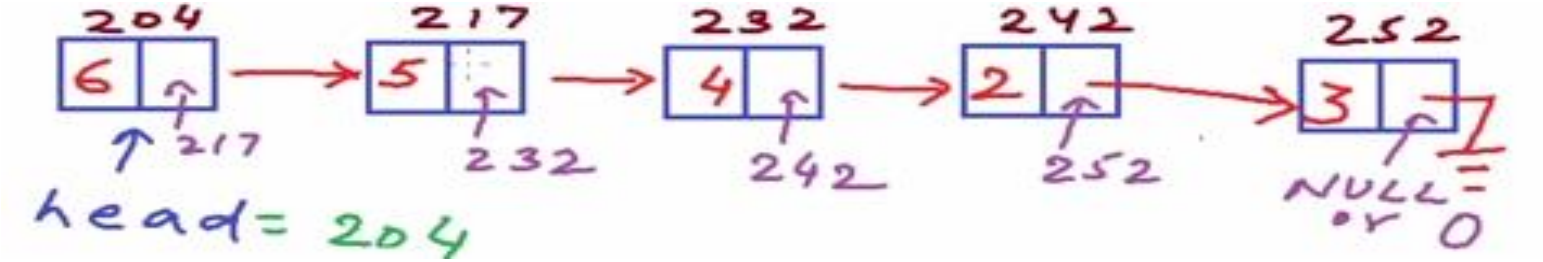
Bağlı Liste – Çözüm (devam…)

- Bağlı listenin hafıza organizasyonu ve yerleşimi yandaki gibi de olabilir.



Bağlı Liste – Çözüm (devam…)

- Listenin **sonuna** "3" sayısını eklersek,
- Yeni eleman NULL'a işaret ederken, yani ***listenin son elemanı*** olurken,
- 2 artık "252" ye (3'ün adresine) işaret eder.
- Aşağıdaki şekilde dizinin bağlı liste yapısı verilmiştir.

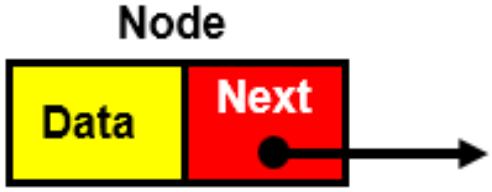


Bağlı Liste – Özet

- Bir bağlı liste, **elemanlarının bir sonraki (Next)** elemanın **hafızadaki yerine** işaret ettiği **zincir şeklinde** bir veri yapısıdır.
- 2 tip bağlı liste bulunmaktadır.
  - **Tek yönlü bağlı liste**
  - **Çift yönlü bağlı liste**
    - **Her iki tipin ayrıca Dairesel bağlı liste oluşumu mevcuttur**

Bağlı Liste – Özet (devam…)

- Bağlı listede her bir **elemana düğüm (node)** adı verilir.
- Her **düğüm**
  - **Data: Veri** (bir **tamsayı**, bir **dizi**, bir **nesne** olabilir)
  - **Next:** ***Sonraki verinin adresine*** **işaret eden bir referanstan** oluşur.




```
struct Node
{
        int Data;
        Node Next;
}
```

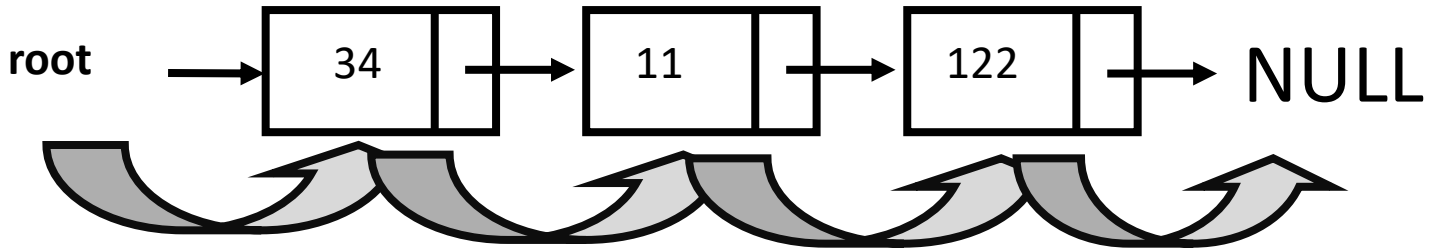
Bağlı Liste – Özet (devam…)

- **İlk düğümün** adresini içeren ve **başlangıç (root)** olarak adlandırılan bir ***referans*** değişkeni bulunur.
  - Bir listeyi gezmek için bu adrese mutlaka ihtiyaç vardır.
  - Boş bir listede **root** **NULL**'a işaret eder.
- Bağlı listede, **son düğümde** **listenin sonuna işaret eden** özel **NULL** (veya sıfır) değeri bulunur.

Bağlı Liste – Gezinme

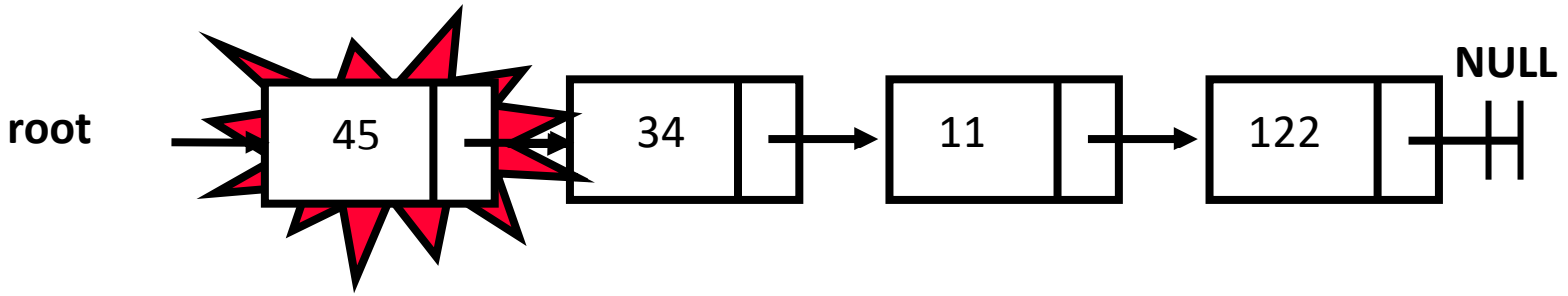
- root düğümünden başlayarak, sonuncu düğüme kadar doğrusal şekilde nasıl ilerleriz? Algoritma, mantık?

```
Node *temp = root;
while (temp != null)
{
        temp = temp->Next;
}
```



Bağlı Liste – InsertFirst

- Listenin ilk elemanı olan **root'u** yeni gelen elemanla yer değiştirdiğimiz metottur.
- **Traverse** işlemine **gerek yoktur**.
- Aşağıdaki listede *root 34'tür*.
- Listenin başına 45 elemanını eklemek istiyoruz. Yani yeni root 45 olmalı. **Nasıl?**



Bağlı Liste – InsertFirst (devam…)

- **Sözde kod:**

a) Yeni root düğüm oluşturulur (**yrd**),

b) root NULL ise (**Liste BOŞSA**)

- `root = yrd`

c) != Null ise Yeni düğümün işaretçisi root'a işaret eder,

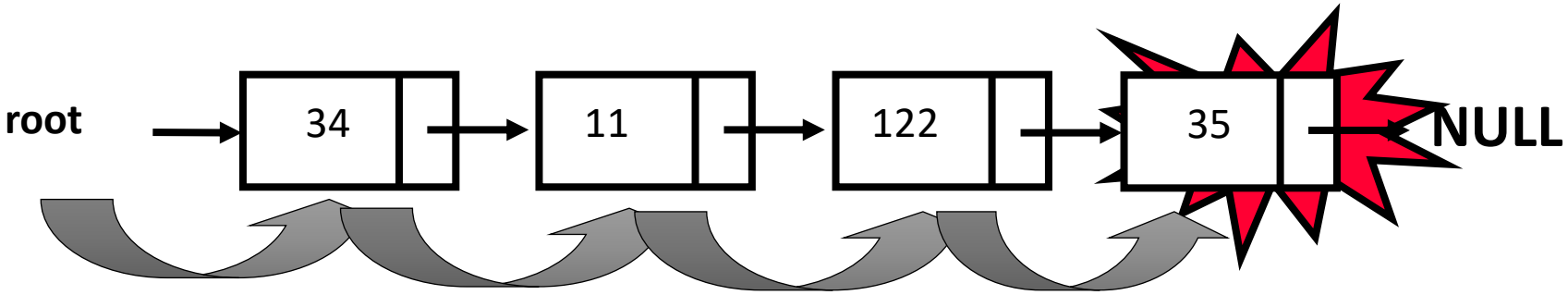
- `yrd->next = root`

root yeni düğüme işaret eder.

- `root = yrd`
- `yrd = null'a işaret edilip silinir.`

Bağlı Liste – InsertLast

- Listenin son elemanını yeni gelen elemanla yer değiştirdiğimiz metottur.
- **Traverse** işlemi **gereklidir**. **Eski sonuncu elemanın** *bulunması* gerekmektedir.
- Aşağıdaki listede *Son eleman 122'dir*.
- Listenin sonuna 35 elemanını eklemek istiyoruz. **Nasıl?**



Bağlı Liste – InsertLast (devam…)

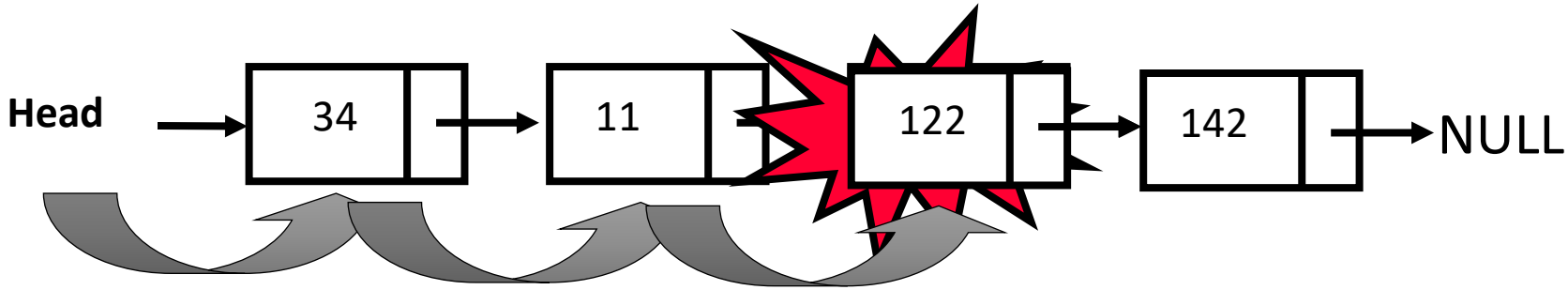
- **Sözde kod:**

a) Yeni işaretçi oluşturulur (**yrd**),
b) root NULL ise (**Liste BOŞSA**)
  - `InsertFirst()`

c) "Eski son düğüm" bulunur (**yrd->next == null**),
d) son düğüme, yeni bir düğüm eklenir.
  - `yrd->Next = new node`

Bağlı Liste – InsertPos

- Listenin **i. pozisyonundaki** elemanın sonrasına yeni gelen elemanı, **ekleyen** metottur.
- **Traverse** işlemi **gereklidir**. **i. Pozisyona sahip elemanın** *bulunması* gerekmektedir.
- Aşağıdaki listede ***i=2 için 11 elemanı ile 142 elemanı arasına*** *122 elemanını ekleyelim*. **Nasıl?**



Bağlı Liste – InsertPos (devam…)

- **Sözde kod:**

  a) Yeni düğüm oluşturulur (**temp = new node**),

  b) root NULL ise (**Liste BOŞSA**)

  - `InsertFirst()`

  c) "i. Pos düğüm" bulunur (**yrd**),

  d) temp elemanının Next'i **yrdNext'e** atanır.

  - `temp->Next = yrd->Next`

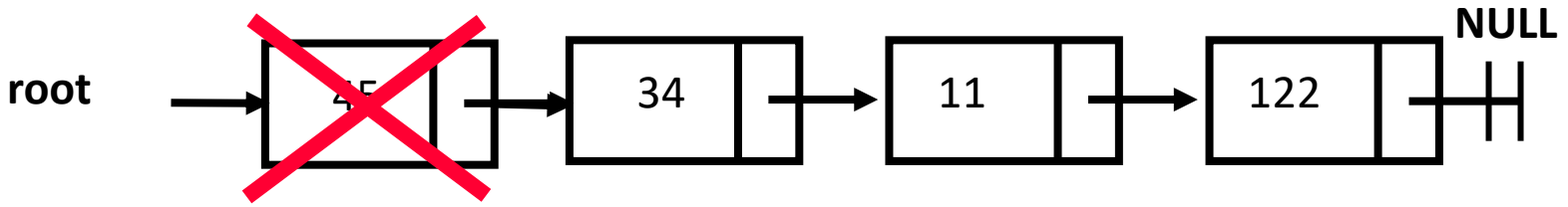
  e) yrd pointer'ın Next'i **temp** pointer'e tanır.

  - `yrd->next = temp`

  f) Kullanılan destek işaretçiler silinir.

Bağlı Liste – DeleteFirst

- Listenin ilk elemanı olan **root'i** silen metottur.
- **Traverse** işlemine **gerek yoktur**.
- Aşağıdaki listede root *45'dir*.
- Listenin başındaki elemanı silmek istiyoruz.
- Yeni root **34 olmalı**. **Nasıl?**



Bağlı Liste – DeleteFirst (devam…)

- **Sözde kod:**

a) root'un işaret ettiği elemana yrd işaretçi değişkeni atanır.

- `yrd = root`

b) `root->next NULL ise` root **NULL** olur,

- **Liste boşalır**

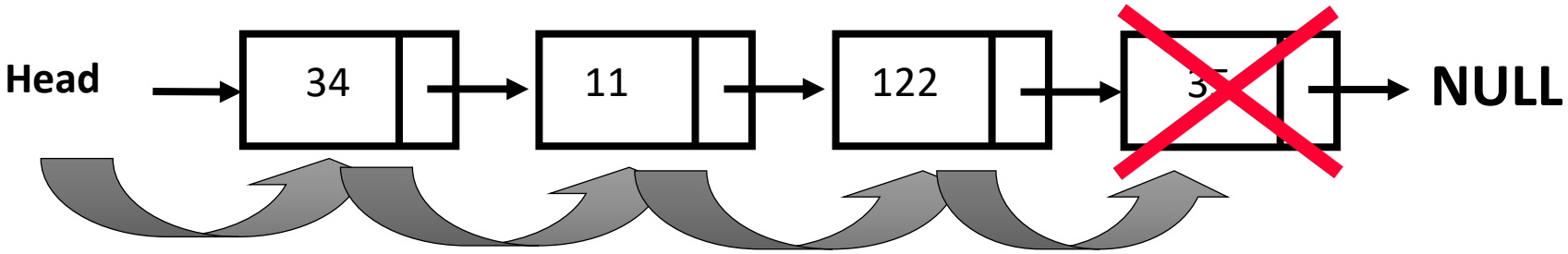
c) `rootNext NULL değilse` root elemanı **yrd->**`Next` olur,

- `root = yrd->next`

d) `yrd node kullanılır ve silinir.`

Bağlı Liste – DeleteLast

- Listenin son elemanını silen metottur.
- **Traverse** işlemi **gereklidir**. **Sonuncu elemanın** *bulunması* gerekmektedir.
- Aşağıdaki listede *Son eleman 35'dir*.
- Son eleman silindiğinde **yeni son eleman** 122 olmalıdır. **Nasıl?**



Bağlı Liste – DeleteLast (devam…)

- **Sözde kod:**

a) Son eleman bulunması için root'tan itibaren doğrusal ilerlenir.
   - `yrd->next->next = Null`

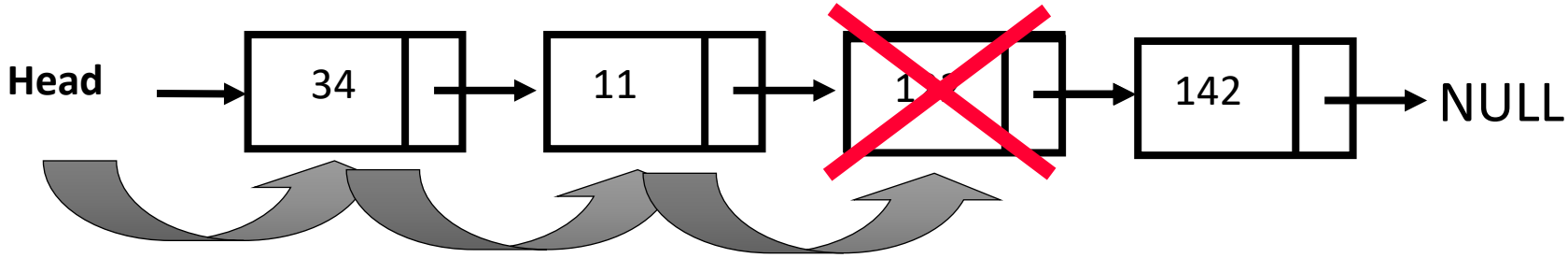
b) Böylece son elemana ulaşmak için sondan bir önceki elemanı gösteren bir `yrd işaretçi` atanmış olur.

c) `Bir temp işaretçi yrd nextine işaret ettirilir` **ve yrd nexti null yapılarak temp listeden koparılır.**
   - `yrd->next = null`

d) `Temp` kullanılarak **silinir**.

Bağlı Liste – DeletePos

- Listenin **i. pozisyonundaki** elemanını **silen** metottur.
- **Traverse** işlemi **gereklidir**. **i. Pozisyona sahip elemanın** *bulunması* gerekmektedir.
- Aşağıdaki listede ***i=2 için 122 elemanını*** *silelim ve 11* elemanı artık **122'yi değil, 142'yi işaret etsin**. **Nasıl?**



Bağlı Liste – DeletePos (devam…)

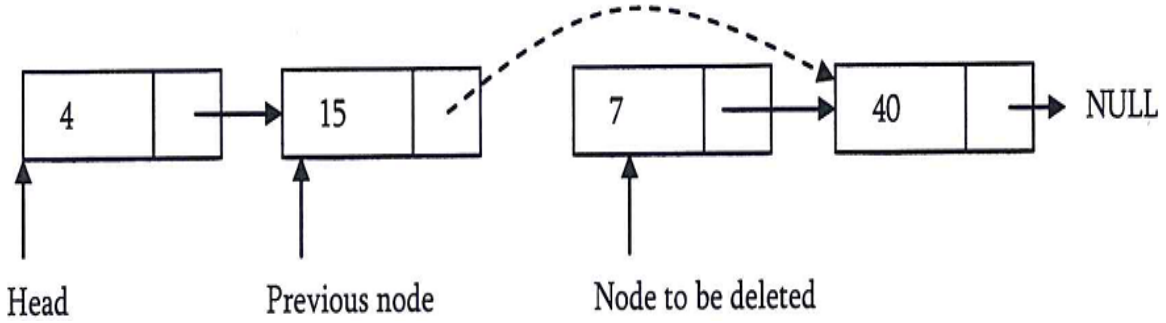
- **Sözde kod:**

a) i. Pozisyondaki elemanın bulunması için root'tan itibaren doğrusal ilerlenir.

b) i. Pozisyonundaki eleman `pos` değişkenine **atanır**.

c) i. Pozisyonundaki eleman bulunduğunda, **ondan bir önceki elemanın Next'ine pos'un Next'i atanır**.

`d) pos` değişkenine **NULL** atanır.



Bağlı Liste ve Dizi Karşılaştırması

- **Erişim Maliyeti**
  - Dizinin 1. veya en sonuncu elemanına erişmek için sadece base adresle **ilgili elemanın sırasının** byte değeri (6. sıra * 4 byte) toplanır. Bu her eleman için sabit bir zamandır ve karmaşıklığı **O(1)** olur.
  - Bağlı liste, **sürekli bir adreste tutulmadığından** en kötü durumda (**worst-case**) son elemanın adresine erişmek için Head elemanından başlanarak **sırayla** her düğümden *bir sonraki adres elde* edilir. Bu nedenle karmaşıklık eleman sayısı n kadar ya da **O(n)** olur.

Bağlı Liste ve Dizi Karşılaştırması (devam..)

- **Veri Giriş (Insert) Maliyeti**
  - **En Başa:** Dizide tüm elemanlar bir kayar **O(n)**. Bağlı listede sabit bir işlem **O(1)**.
  - **En Sona:** *Dizi* **boşsa O(1)** değilse yeni bir dizi gerekeceğinden en kötü O(n). *Bağlı listede* sonuncu elemana **en kötü O(n)** le ulaşılır ve eklenir.
  - **i. Pozisyona:** *Dizide* ilgili elemana ulaşmak ve duruma göre elemanları kaydırmak gerekeceği için **en kötü durumda O(n)** zamana ihtiyaç vardır. *Bağlı listede* ise kaydırma olmadığı halde bir elemana erişmek için head adresinden istenen adrese kadar gezmek gerekeceğinden **en kötü durumda O(n)** olur.

Bağlı Liste ve Dizi Karşılaştırması (devam..)

- **Silme (Delete) Maliyeti**
  - Insert işlemleri ile tamamen aynı senaryolara ve aynı Big-O karmaşıklık maliyetlerine sahiptirler.

Bağlı Liste ve Dizi Karşılaştırması (devam..)

İşlem	Dizi Tabanlı Liste	Bağlı Liste
Access, Retrieve (Erişim)	O(1)	O(n)
InsertFirst	O(n)	O(1)
InsertLast	O(n)	O(n) Sonuncu elemana pointer varsa = O(1)
InsertPos	O(n)	O(n)
DeleteFirst	O(n)	O(1)
DeleteLast	O(n)	O(n) Sonuncu elemana pointer varsa = O(1)
DeletePos	O(n)	O(n)

İYİ ÇALIŞMALAR…

Yararlanılan Kaynaklar

- **Sunum**
  - Yrd. Doç. Dr. Deniz KILINÇ (Celal Bayar Üniversitesi)
- **Ders Kitabı:**
  - Data Structures through JAVA, V.V.Muniswamy
- **Yardımcı Okumalar:**
  - Data Structures and Algorithms in Java, Narashima Karumanchi
  - Data Structures, Algorithms and Applications in Java, Sartaj Sahni
  - Algorithms, Robert Sedgewick