

# **BLM212 Veri Yapıları**

## **Linear Lists (Doğrusal Listeler)**



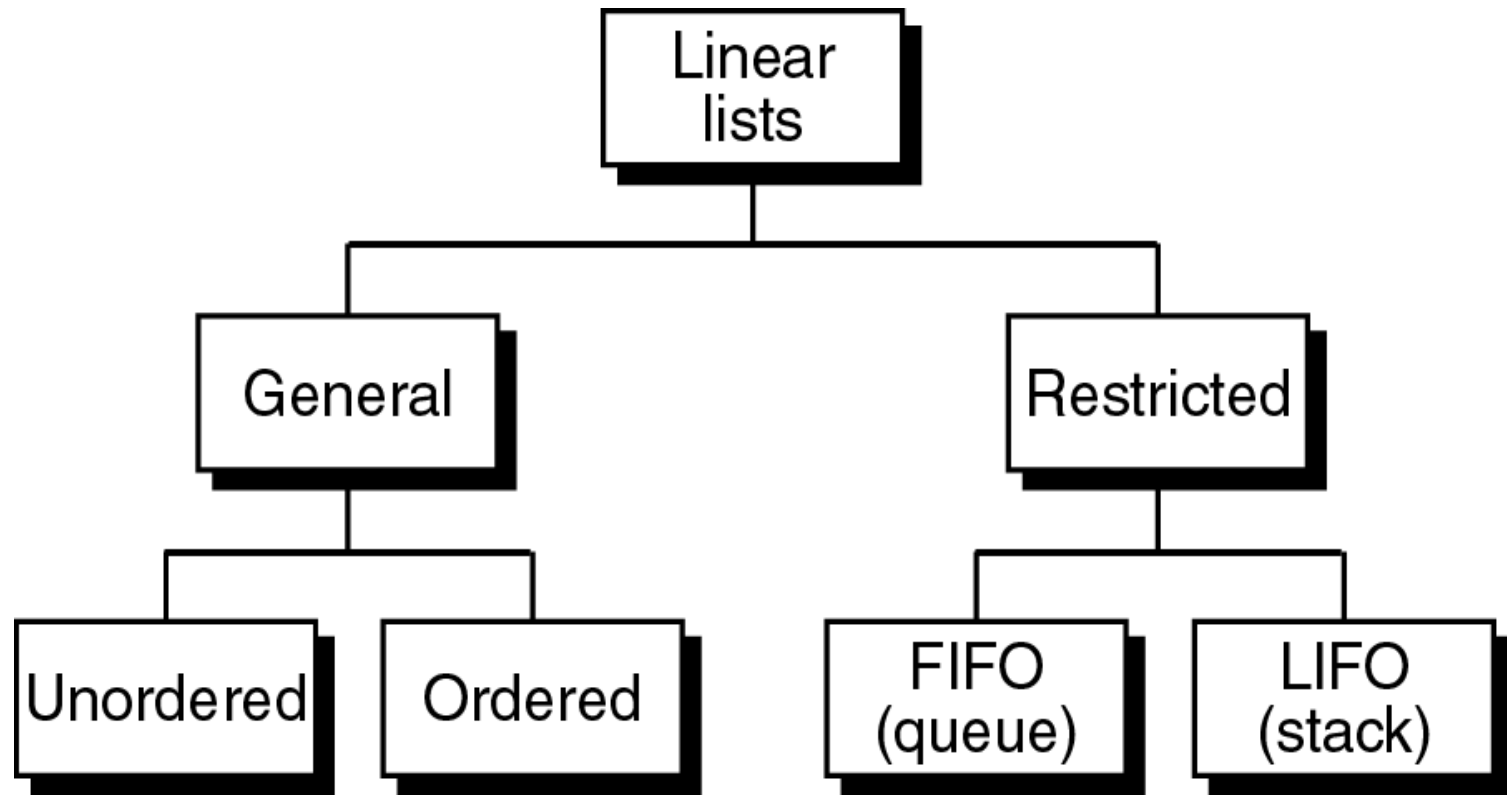
# *Linear List*

## Hedefler

---

- Doğrusal bir listenin tasarımını, kullanımını ve çalışması açıklamak
- Bağlı liste (linked-list) yapısı kullanarak doğrusal bir liste uygulamak/gerçekleştirmek
- Linear List ADT nin çalışmasını anlamak
- Linear List ADT kullanarak uygulama programlarını yazmak
- Farklı bağlı liste yapıları tasarlamak ve uygulamak

# Linear Lists



Operations are;

1. Insertion
2. Deletion
3. Retrieval
4. Traversal (exception for restricted lists).

# Temel Operasyonlar

- **Insertion** (Ekleme)
- **Deletion** (Silme)
- **Retrieval** (Erişme/Alma)
- **Traversal** (Gezinme)

# Linear Lists



# Linear Lists

## Insertion

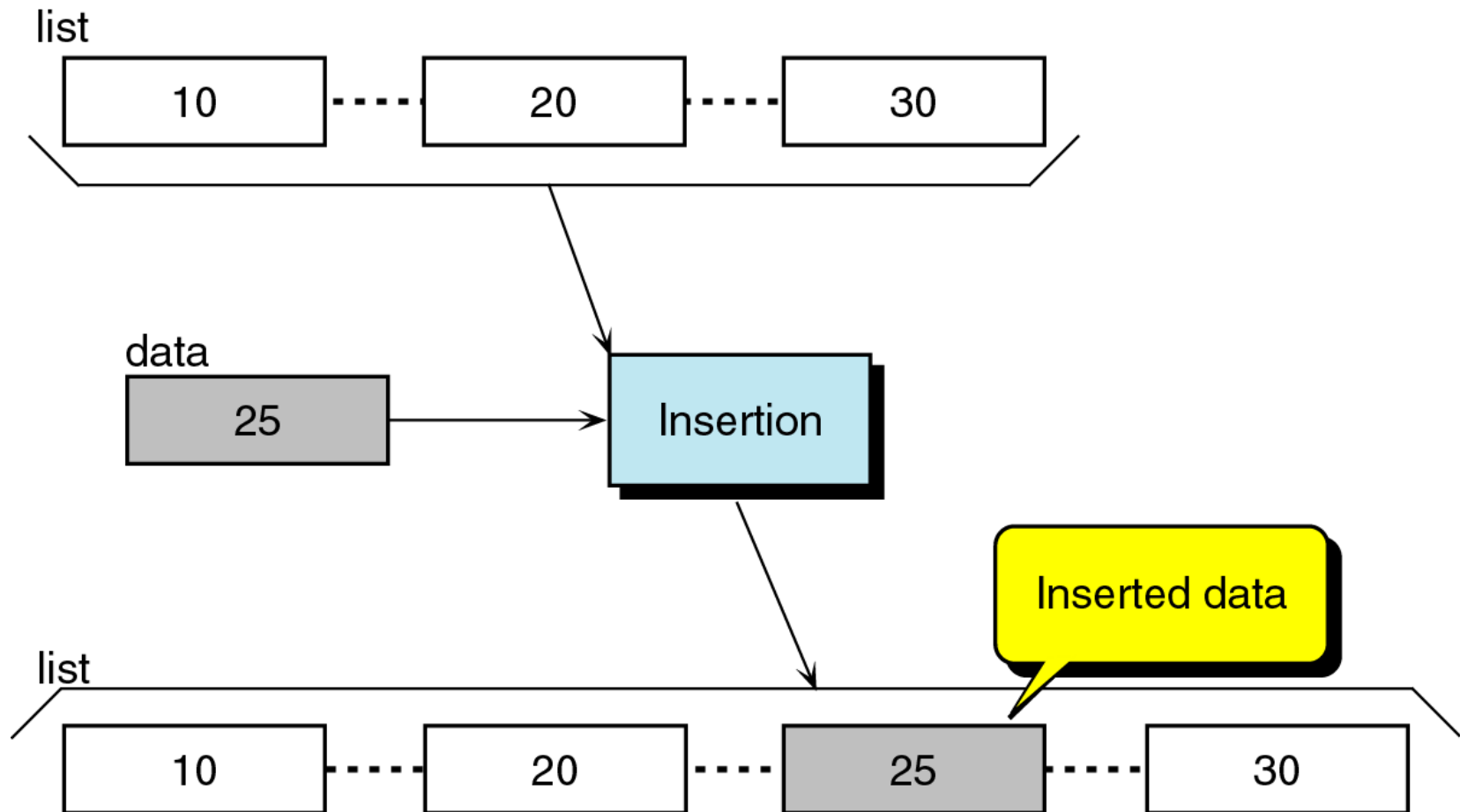


Figure 3-3

# Linear Lists

## Deletion

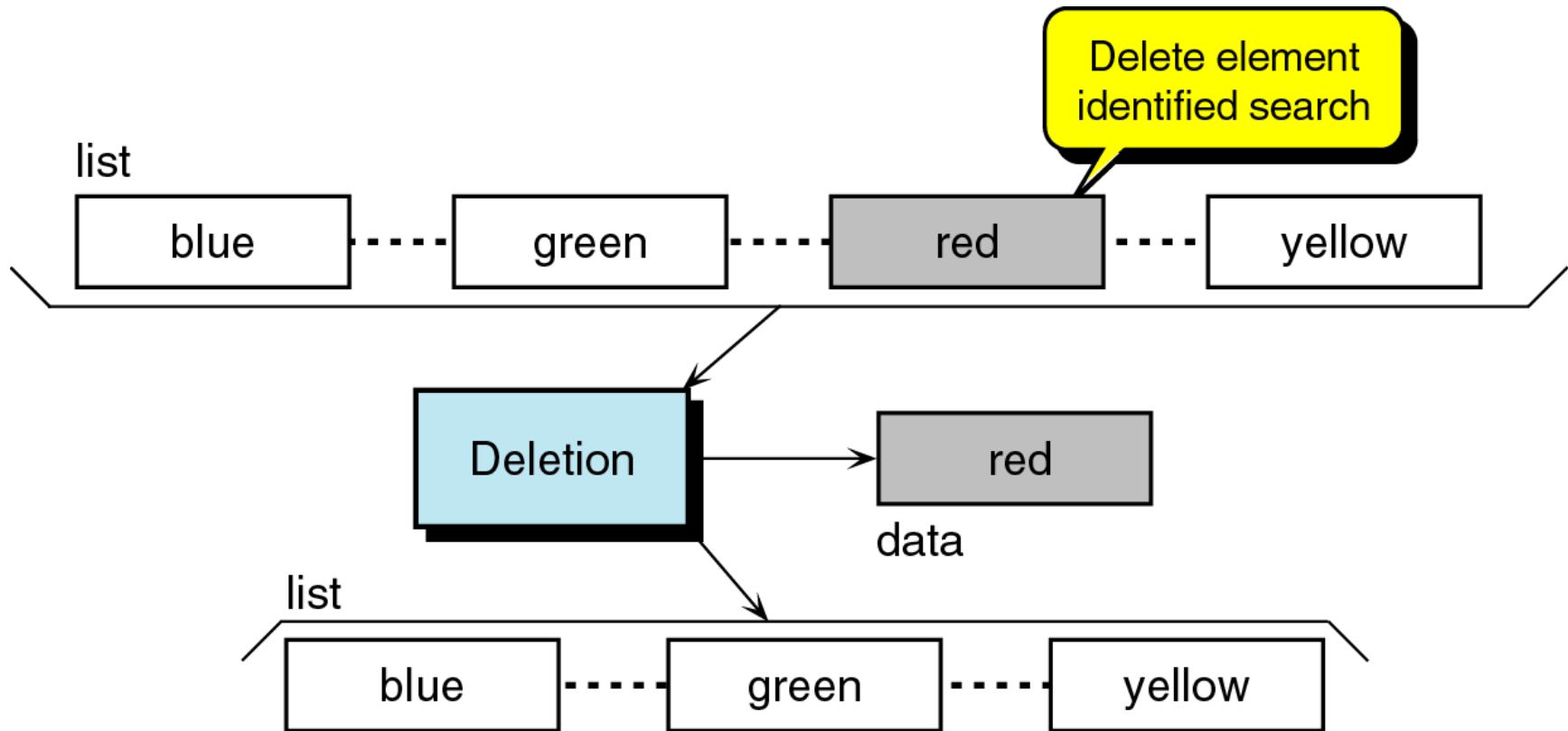


Figure 3-4

# Linear Lists

## Retrieval

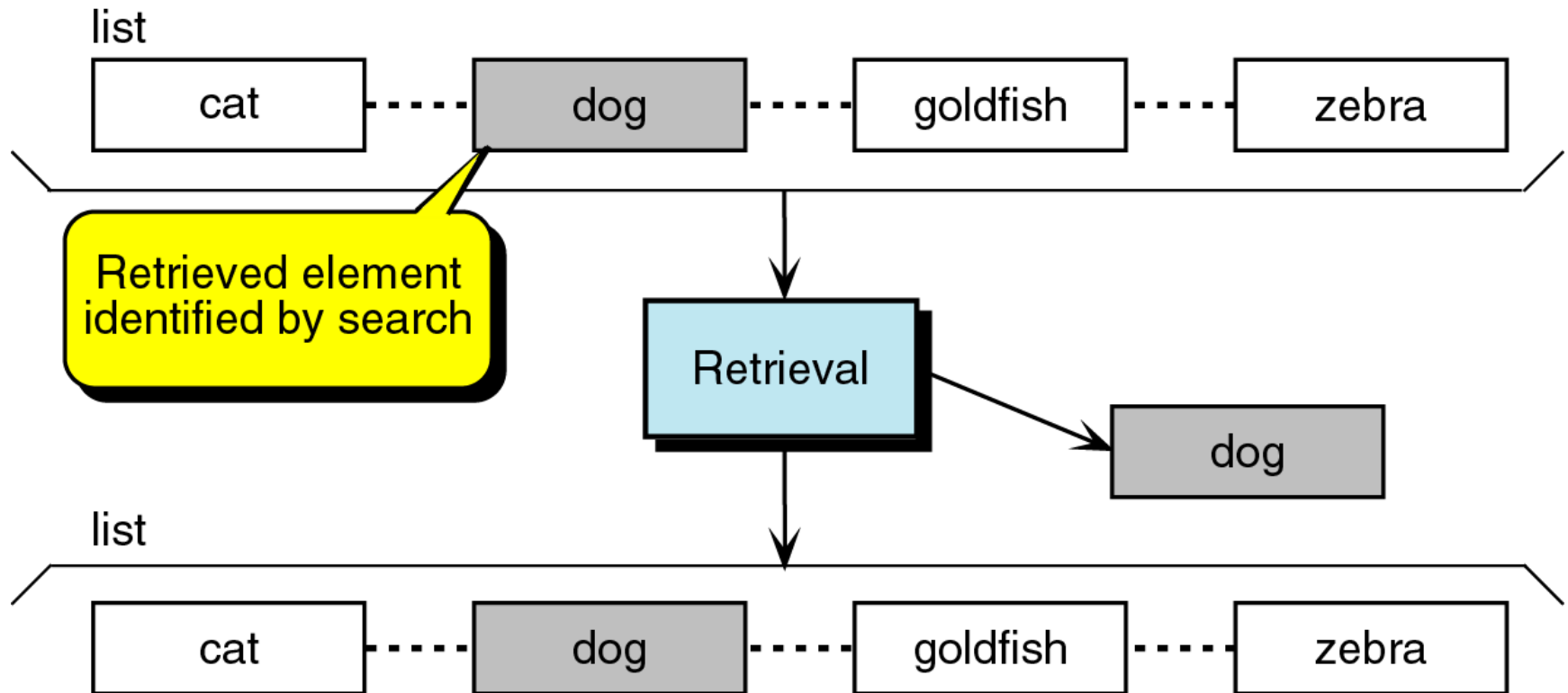


Figure 3-5



# Implementation

- **Array Implementation of Lists**
- **Linked Lists**

# Linear Lists

- Sıralı verilerin **eklenmesi** veya **silinmesi** gerektiğinde diziler (arrays) verimsizdir.
- Bağlı listede ise ekleme (**insertion**) ve silme (**deletion**) işlemleri **etkin** bir şekilde gerçekleştirir.
- Fakat arama (**search**) ve erişme (**retrieval**) **verimsizdir**.

# Array Implementation of Lists

N elemanlı bir

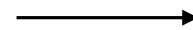
- Listenin **başına** eleman eklemek/silmek



Complexity

$O(N)$   
Linear time

- Listenin **sonuna** eleman eklemek/silmek



$O(1)$   
Constant time

- Listenin tümünü yazdırmak (**printList**)



$O(N)$   
Linear time

- Elemana erişme (**findKth** operation)



$O(1)$   
Constant time

Eğer uygulamanız sadece listenin sonuna ekleme gerektiriyor ise sadece diziye erişim (yani **findKth** operations) meydana gelir. Bu durumda **dizi implementasyonu** uygundur. Eğer ekleme ve silmeler listenin her noktasında özellikle de başında meydana geliyorsa **array** iyi bir seçim değildir.

# Linked Lists

N elemanlı bir

- Listenin **başına** eleman eklemek/silmek

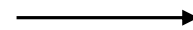


Complexity

$O(1)$

Constant time

- Listenin **sonuna** eleman eklemek/silmek



$O(1)$

Constant time

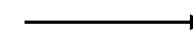
- Listenin tümünü yazdırmak (**printList**)



$O(N)$

Linear time

- Elemana erişme (**findKth** operation)

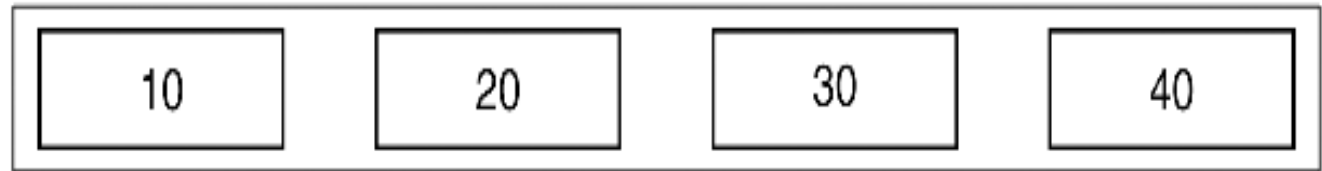


$O(i)$

**findKth** işlemi artık bir dizi implementasyonundaki kadar verimli değildir. Bu işlem için listeyi gezinmek (traverse) gerekir ve bu  $O(i)$  zaman alır.

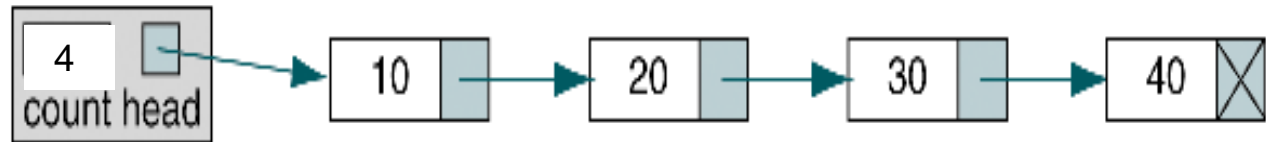
# Linked Lists

List



(a) Conceptual view of a list

List



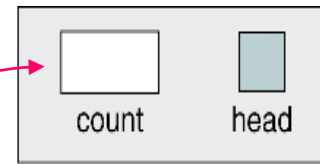
(b) Linked list implementation

FIGURE 5-4 Linked List Implementation of a List

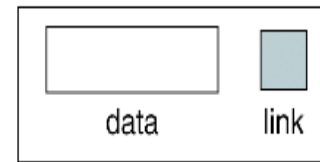
- Bağlı/Bağlantılı liste, liste oluşturmak için iyi bir yapıdır; çünkü veriler listenin başına, ortasına veya sonuna **kolayca eklenir ve silinir**.

Listeyi tanımlamak için yalnızca tek bir **işaretçi** gerekli olmasına rağmen, genellikle baş işaretçi ve listeye ilgili diğer verileri depolayan bir baş düğüm yapısı (**head structure**) oluşturmak uygundur.

Bir düğüm bir liste hakkında veri içerdiğinde, o veriler **metadata** olarak anılır; yani, listedeki verilerle ilgili verilerdir.



(a) Head structure



(b) Data node structure

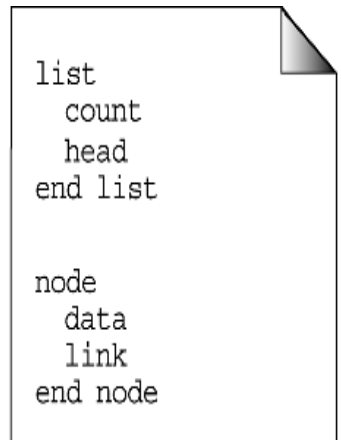
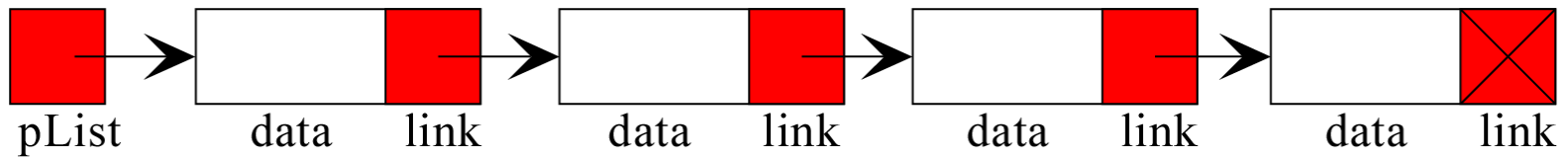


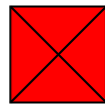
FIGURE 5-5 Head Node and Data Node

Başka **metadata** örneği?

# Linked Lists



(a) A linked list with a head pointer: `pList`



`pList`

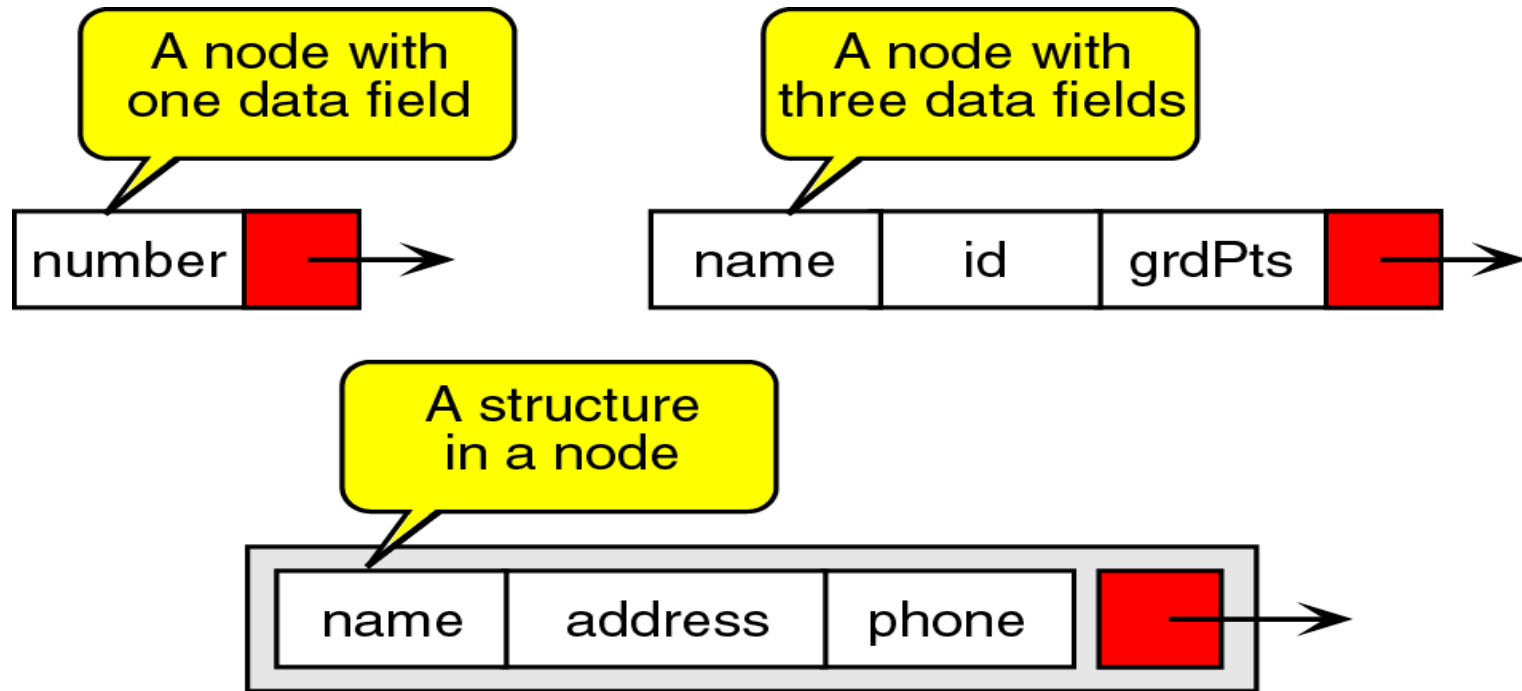
(b) An empty linked list

# Linked Lists

- Bağlı listelerin (linked lists) avantajları;
  - Veriler kolayca eklenir veya silinir. Bağlı listede elemanları kaydırmaya gerek yoktur.
- Bağlı listelerin dezavantajları;
  - Sıralı aramalarla sınırlıdır

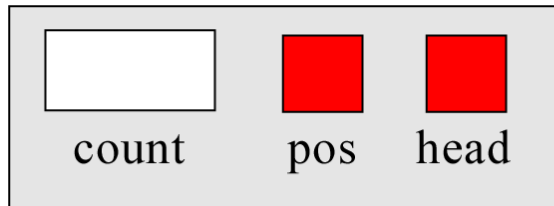


# Linked Lists

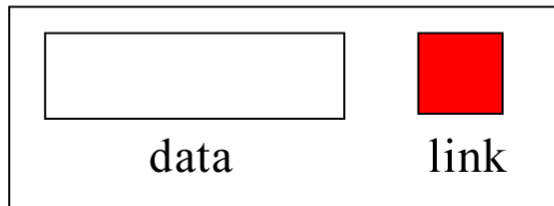


- Bağlı listedeki elemanlara **düğüm** (**nodes**) denir.
- Bağlı listedeki düğümler **self-referential** yapılar olarak anılır. Bu tür bir yapıda, yapının her bir örneği aynı yapısal türün bir başka örneğine işaret eden pointer içerir.

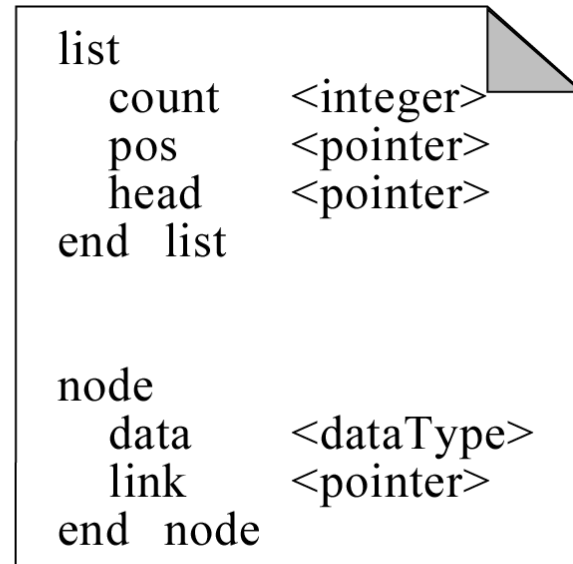
Figure 3-7



(a) head structure



(b) data node structure



**Figure 3-8**

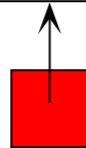
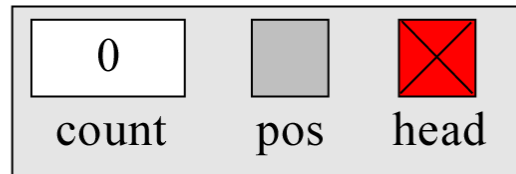
# Create List

(a) Before create



pList

```
pList = createList
```



pList

(b) After create

Figure 3-9

# Create List

## Algorithm **createList**

Allocate dynamic memory for a linked list head node and returns its address to caller.

PRE Nothing

POST Head node allocated or error returned

RETURN head node pointer or null if memory overflow

1. if (memory available)
    1. allocate (pNew)
    2. pNew → head = null pointer
    3. pNew → count = 0
  2. else
    1. pNew = null pointer
  3. return pNew
- end **createList**

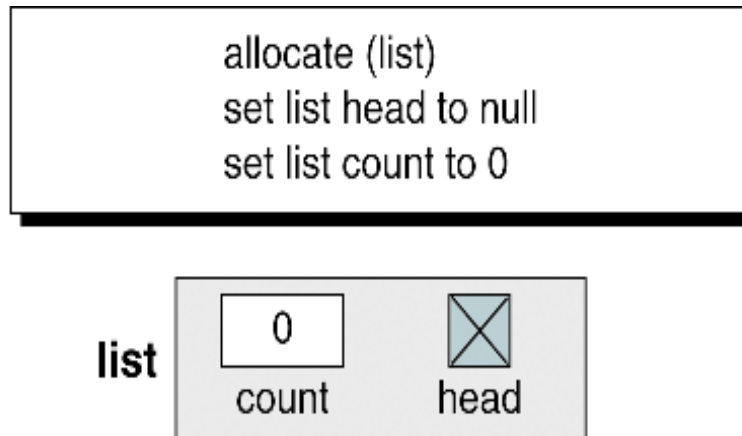


FIGURE 5-6 Create List

#### ALGORITHM 5-1 Create List

```
Algorithm createList (list)
Initializes metadata for list.
  Pre   list is metadata structure passed by reference
  Post  metadata initialized
1 allocate (list)
2 set list head to null
3 set list count to 0
end createList
```

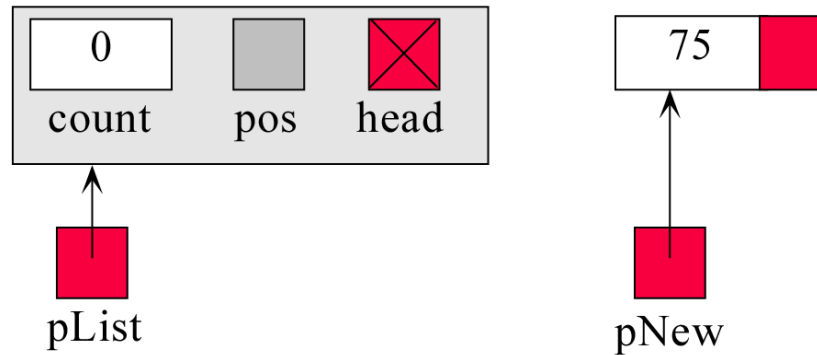
# Insert Node

- Bağlı listeye eleman ekler. Aşağıdaki 3 adımda gerçekleştirilir.
  1. Yeni düğüm için bellek tahsis edilir ve veri ona eklenir.
  2. Yeni düğümden önceki düğümün yerini bilmemiz gerekir.
    - ❑ Eğer kendinden önceki **null** ise, eklenen elemandan önce gelen yok demektir. Yani boş listeye veya listenin başına ekleme durumu
    - ❑ Aksi takdirde, listenin ortasına veya sonuna eleman ekleme durumudur
  3. Yeni elemandan önce gelen (predecessor) elemanın yeni elemana işaret etmesi sağlanır.

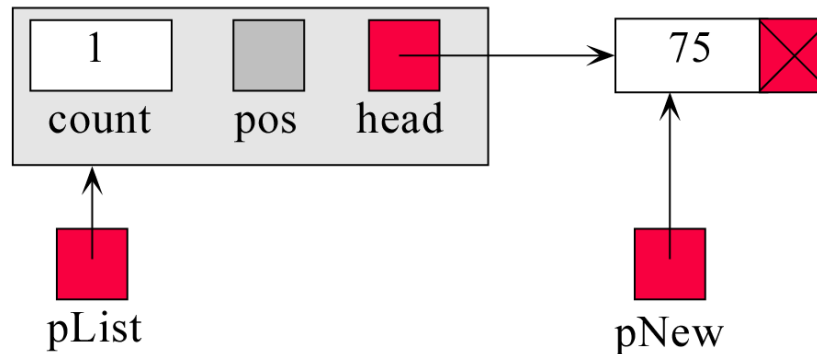
# Insert Node

**Boş listeye eleman ekleme (Add node to empty list)**

(a) Before add



$$\begin{aligned} \text{pNew} \rightarrow \text{link} &= \text{pList} \rightarrow \text{head} \\ \text{pList} \rightarrow \text{head} &= \text{pNew} \end{aligned}$$

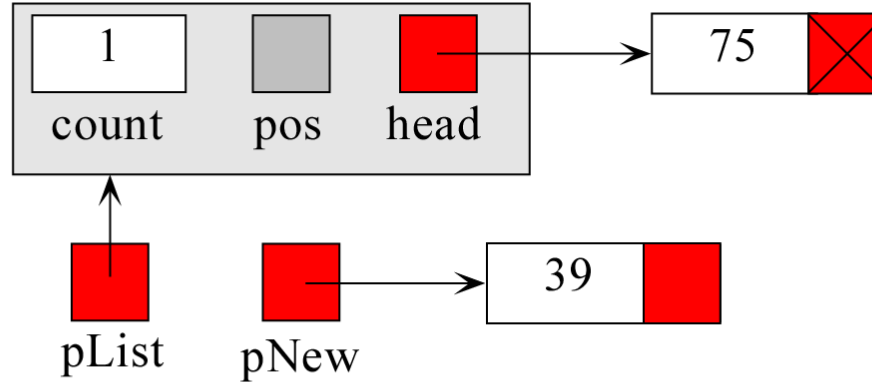


(b) After add

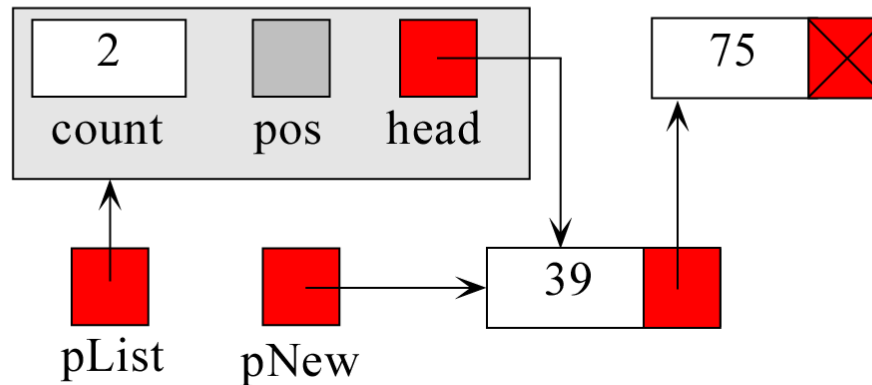
# Insert Node

## Listenin başına eleman ekleme (Add node at beginning)

(a) Before add



$$\begin{aligned} \text{pNew} \rightarrow \text{link} &= \text{pList} \rightarrow \text{head} \\ \text{pList} \rightarrow \text{head} &= \text{pNew} \end{aligned}$$



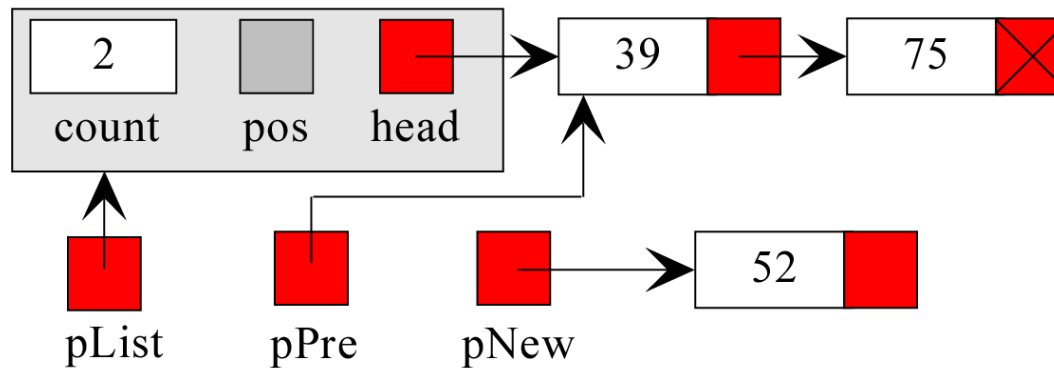
(b) After add



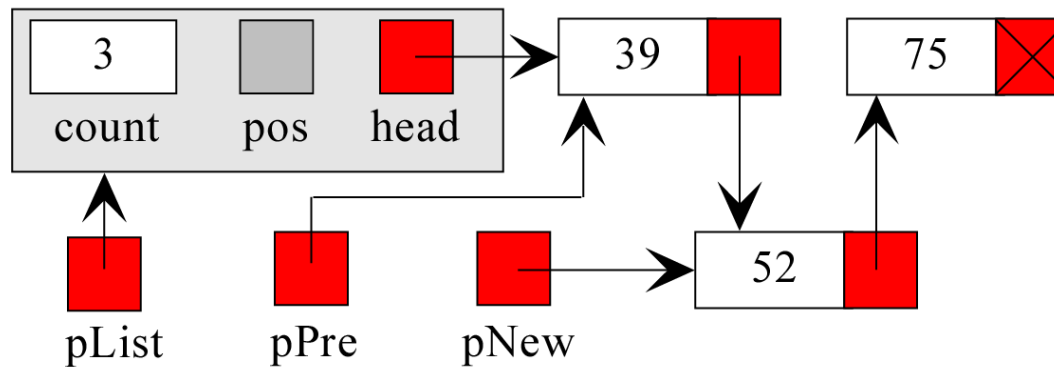
# Insert Node

**Listenin ortasına eleman ekleme (Add node in middle)**

(a) Before add



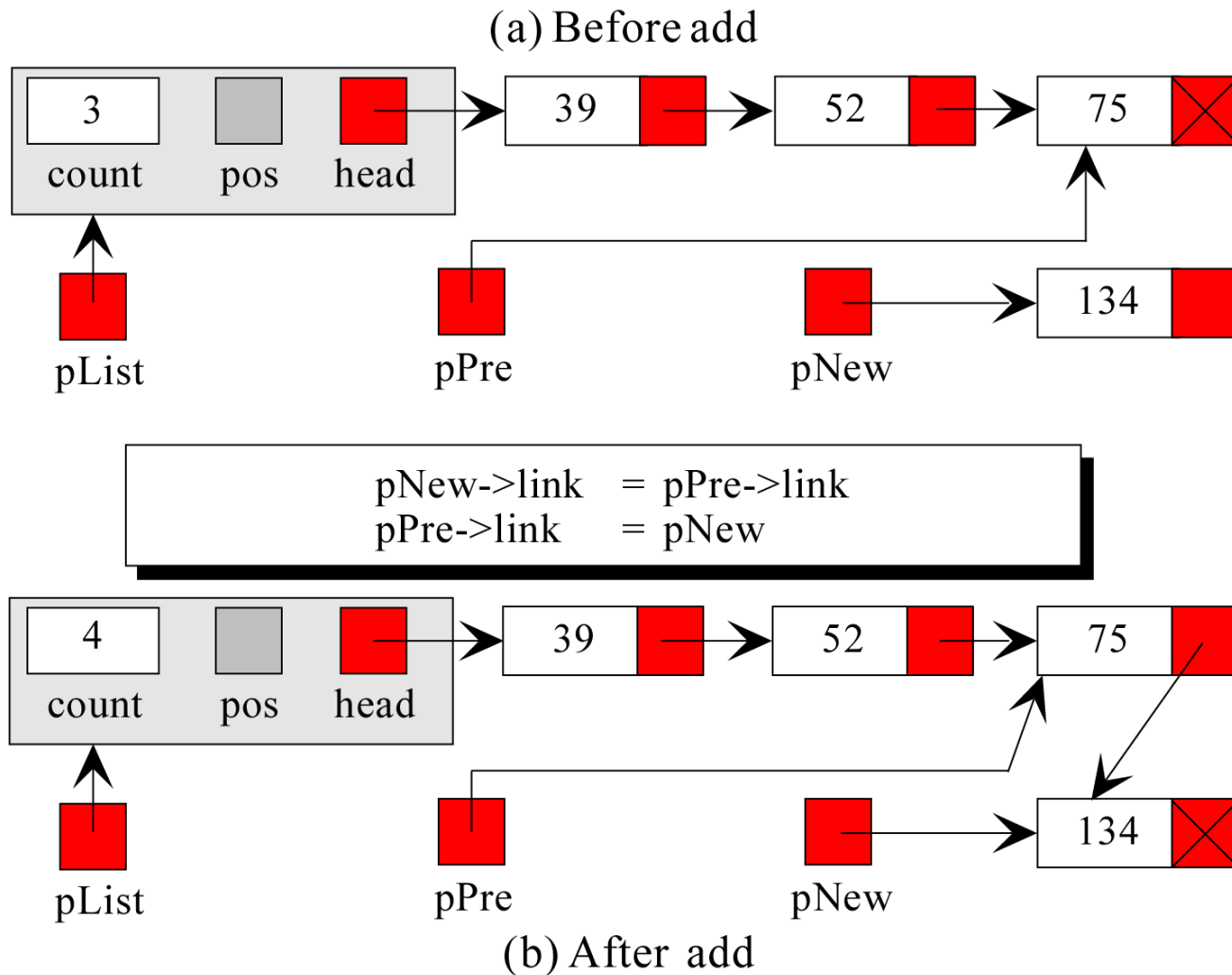
$$\begin{aligned} \text{pNew} \rightarrow \text{link} &= \text{pPre} \rightarrow \text{link} \\ \text{pPre} \rightarrow \text{link} &= \text{pNew} \end{aligned}$$



(b) After add

# Insert Node

Listenin sonuna ekleme (Add node at end)



# Insert Node

Algorithm **insertNode**(val pList <head pointer>, val pPre <node pointer>,  
val dataIn <dataType>)

Insert data into a new node in the linked list.

PRE **pList** is a pointer to a valid list head structure

**pPre** is a pointer to data's logical predecessor

**dataIn** contains data to be inserted

POST data have been inserted in sequence

RETURN true if seccessful, false if memory overflow

# Insert Node

1. allocate (pNew)
2. if (memory overflow) return false
3. pNew→data = dataIn
4. if (pPre null)

Adding before first node or to empty list.

1. pNew→link = pList→head
2. pList→head = pNew

5. else

Adding in middle or at end.

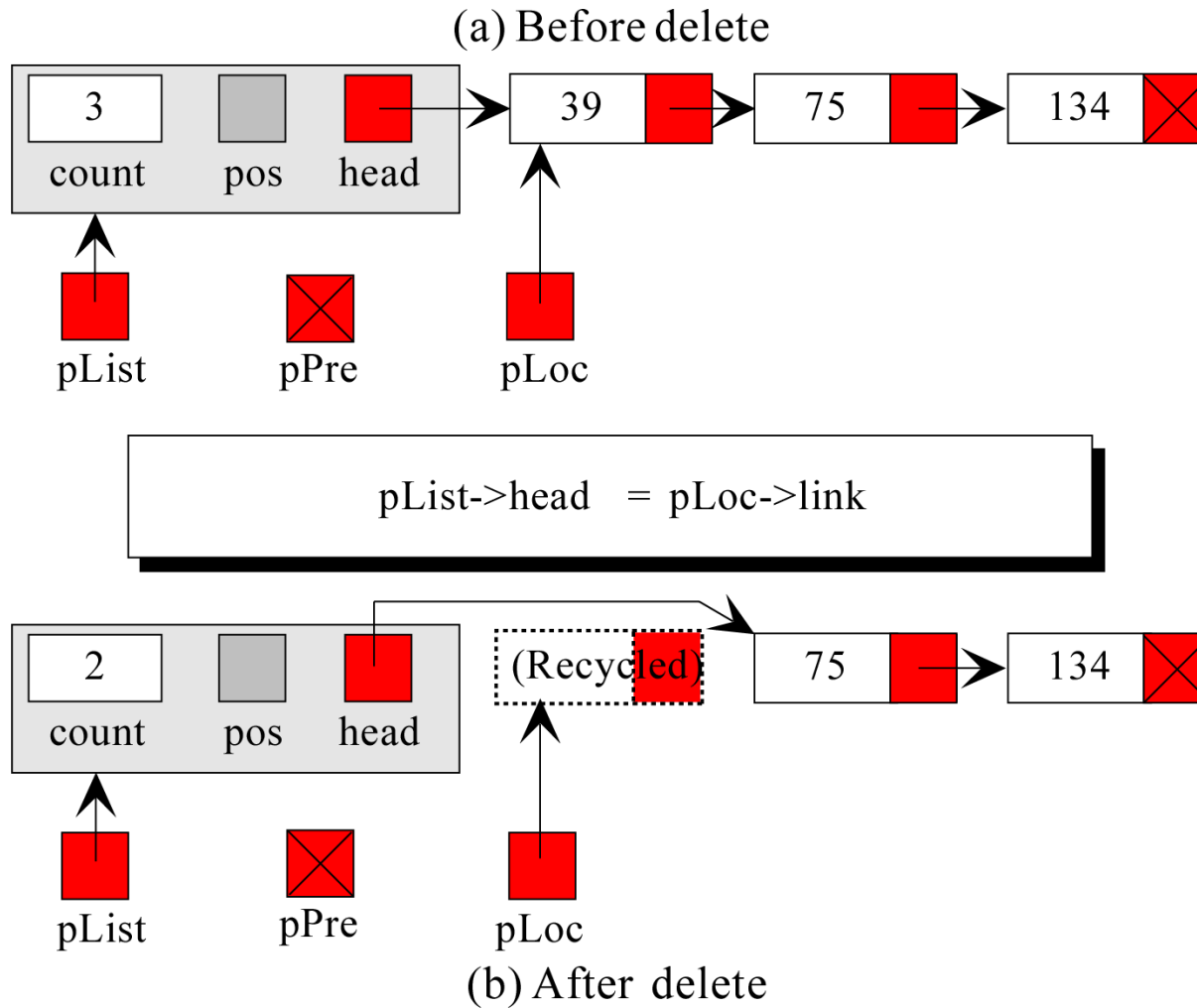
1. pNew→link = pPre→link
2. pPre→link = pNew

6. pList→count = pList→count + 1
7. return true

end **insertNode**

# Delete Node

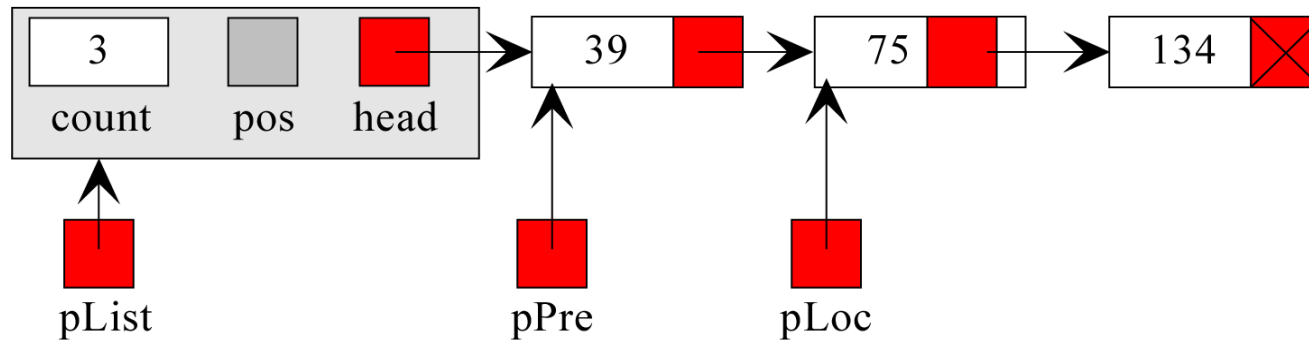
## İlk elemanı silme (Delete first node)



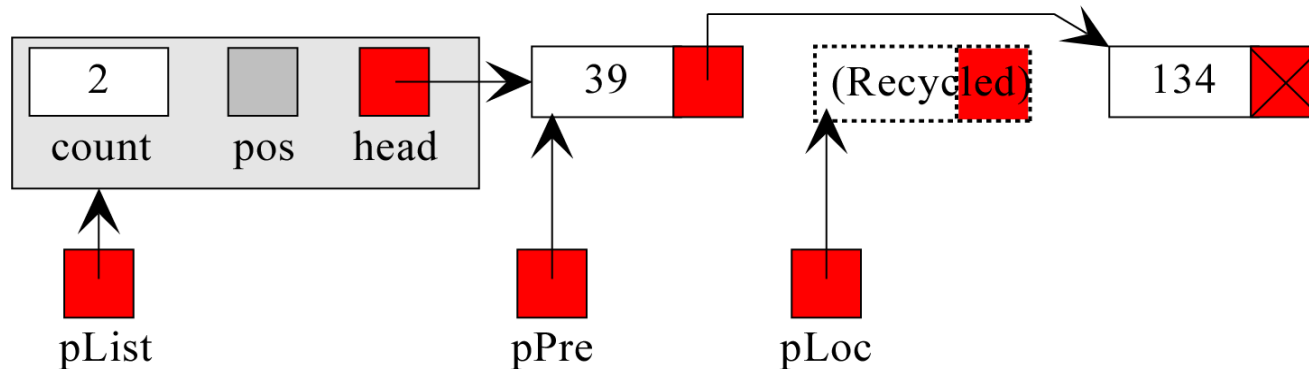
# Delete Node

## Eleman silme (Delete general case)

(a) Before delete



$pPre \rightarrow \text{link} = pLoc \rightarrow \text{link}$



(b) After delete

# Delete Node

Algorithm **deleteNode**(val pList <head pointer>, val pPre <node pointer>,  
val pLoc <node pointer>, ref dataOut <dataType>)

Deletes data from a linked list and returns it to calling module.

PRE **pList** is a pointer to a valid list head structure

**pPre** is a pointer to predecessor node

**pLoc** is a pointer to node to be deleted

**dataOut** is address to pass deleted data to calling module

POST data have been deleted and return to caller

# Delete Node

1. dataOut = pLoc→data

2. if (pPre null)

Deleting first node.

1. pList→head = pLoc→link

3. else

Deleting in middle or at end.

1. pPre→link = pLoc→link

4. pList→count = pList→count – 1

5. release pLoc

6. return

end **deleteNode**



# List Search

- Listeye veri ekleyebilmek için yeni eklenecek veriden (mantıksal) önceki elemanı bilmemiz gerekir.
- Listedden veri silmek için silinecek düğümü bulmamız ve bu veriden (mantıksal) önceki veriyi belirlememiz gerekir.
- Bir veriye erişmek/almak için listede arama yapıp veriyi bulmak gerekir.

# List Search

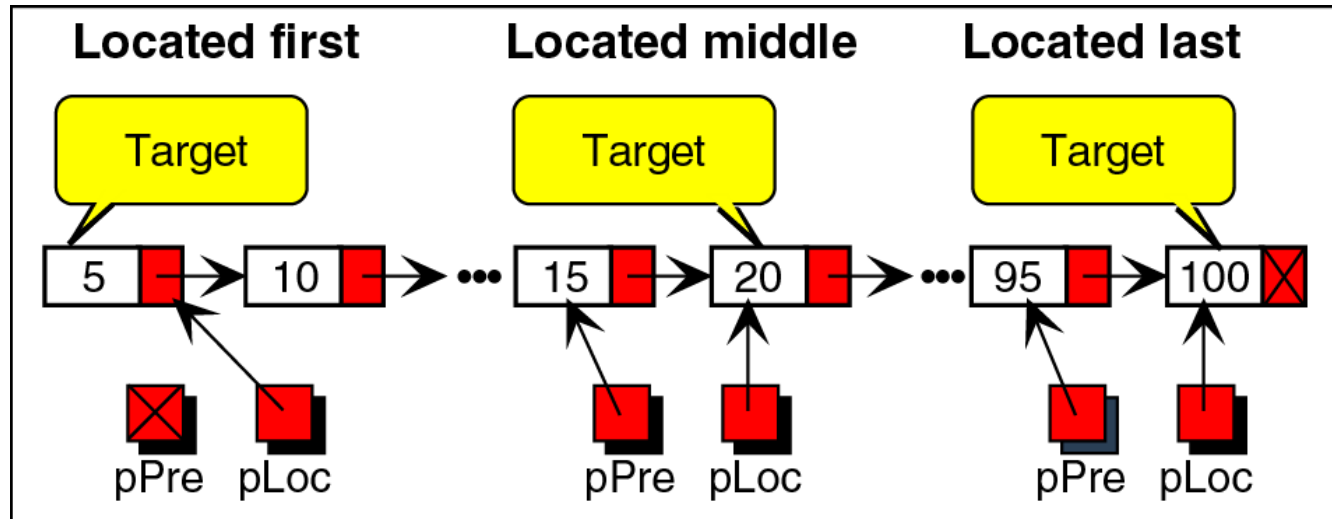
- Sıralı arama (**sequential search**) yapmak zorundayız çünkü düğümler arasında **fiziksel bir ilişki yoktur**.
- Klasik sıralı arama, elemanı listede **bulduğunda** onun konumunu ve **bulamadığında** ise son elemanın adresini döndürür.
- Sıralı liste (ordered list) olduğu için, bulduğunda elemanın konumunu ve bulamadığında ise olması gereken yeri döndürmemiz gerekir.

# List Search

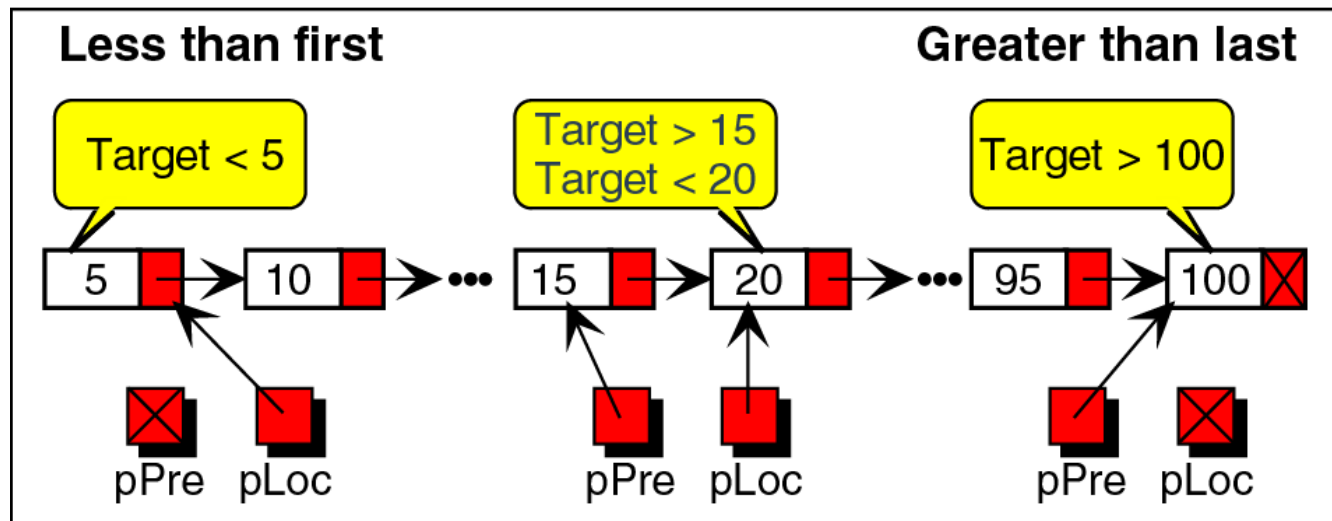
- Bir hedef anahtar (target key) verildiğinde, sıralı liste araması, listedeki istenen düğümü bulmaya çalışır.
- Bir anahtar ile bir listede arama yapmak için bir anahtar alanına (**key field**) ihtiyacımız var.
- Basit listeler için anahtar ve veriler aynı alan olabilir.
- Daha karmaşık yapılar için ayrı bir anahtar alana ihtiyacımız var.
- Listedeki bir düğüm hedef değerle eşleşirse, arama **true** değerini döndürür; Anahtarla eşleşme yoksa **false** döndürür.

```
data
  key
  field1
  field2
  ...
  fieldN
end data
```

# Ordered List Search



(a) Successful searches (return true)

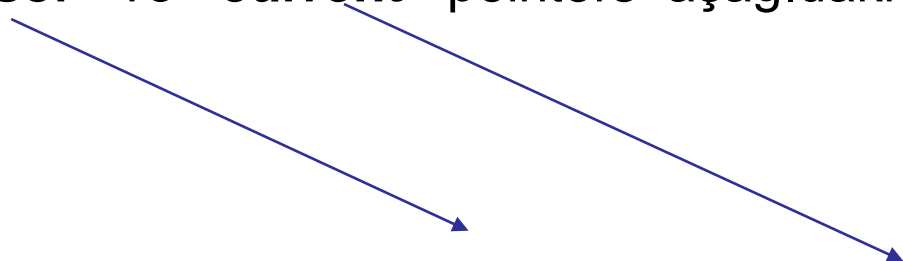


(b) Unsuccessful searches (return false)

Figure 3-16

# List Search

- Listedeki bir düğüm hedef değerle eşleşirse, arama **true** değerini döndürür; Anahtarla eşleşme yoksa **false** döndürür.
- «**Predecessor**» ve «**current**» pointers aşağıdaki tabloya göre setlenir.



Condition	pPre	pLoc	Return
Target < first node	Null	First node	False
Target = first node	Null	First node	True
First < target < last	Largest node < target	First node > target	False
Target = middle node	Node's predecessor	Equal node	True
Target = last node	Last's predecessor	Last node	True
Target > last node	Last node	Null	False

TABLE 5-1 List Search Results

# Search List

Algorithm **searchList**(val pList <head pointer>, val pPre <node pointer>,  
val pLoc <node pointer>, ref target <key type>)

Searches list and passes back address of node containing target.

PRE **pList** is a pointer to a valid list head structure

**pPre** is a pointer variable to receive predecessor node

**pLoc** is a pointer to receive current node

**target** is the key being sought

POST pLoc points to first node with equal or greater than target key

or null if target > key of last node

pPre points to largest node smaller than key

or null if target < key of first node

# Search List

1. pPre = null
2. pLoc = pList→head
3. loop (pLoc not null AND target > pLoc→data.key)
  1. pPre = pLoc
  2. pLoc = pLoc→link

Set return value

4. if (pLoc is null)
  1. found = false
5. else
  1. if (target equal pLoc→data.key)
    1. found = true
  2. else
    1. found = false

6. return found  
end **searchList**

null  
pointer  
test?

İlk koşul  
listenin  
sonunun  
dışına  
çıkmaktan  
korur

İkinci koşul; hedef  
bulunduğunda veya  
hedeften daha  
büyük bir düğüm  
bulunduğunda yani  
hedef, listede  
yoksa döngüyü  
durdurur.

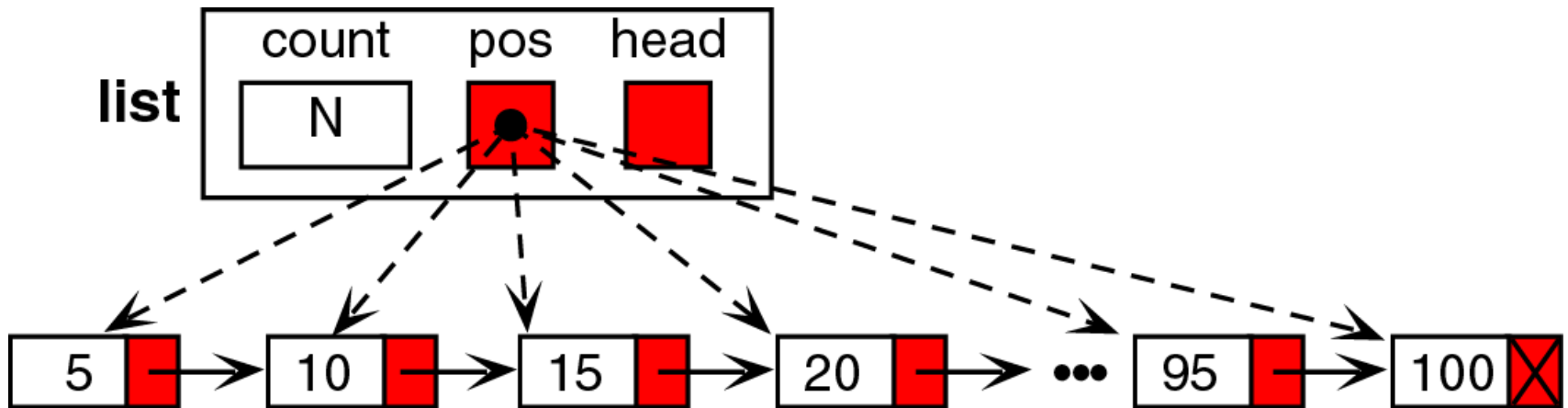
# Traverse List

$\text{pWalker} = \text{pList} \rightarrow \text{head}$

loop (pWalker not null)

process( $\text{pWalker} \rightarrow \text{data}$ )

$\text{pWalker} = \text{pWalker} \rightarrow \text{link}$





# Traverse List

Algorithm **traverse**(val pList <head pointer>, val fromWhere <boolean>,  
ref dataPtr <pointer to dataType>)

Traverses a linked list. Each call returns the location of an element in the list.

PRE **pList** is a pointer to a valid list head structure

**fromWhere** is 0 to start at the first element

**dataPtr** is the address of a pointer to data

POST address places in dataPtr and returns true or if end of list, returns false.

RETURN true if next element located, false if end of list.

# Traverse List

1. if (fromWhere is 0)  
    start from first
    1. if (pList→count is zero) return false
    2. else
      1. pList→pos = pList→head
      2. dataPtr = address(pList→pos→data)
      3. return true
  2. else  
    start from pos
    1. if (pList→pos→link is null)  
        end of list
      1. return false
    2. else
      1. pList→pos = pList→pos→link
      2. dataPtr = address(pList→pos→data)
      3. return true
- end **traverse**

# Destroy List

1. Listedeki düğümleri siler, tahsis edilmiş belleği serbest bırakır.
2. Baş düğümün (head node) işgal ettiği belleği serbest bırakır.
3. Listenin artık mevcut olmadığını gösteren boş göstericiyi geri gönderir

# Destroy List

algorithm **destroyList** (ref pList <head pointer>)

Deletes all data in list and then deletes head structure.

PRE pList is a pointer to a valid list head structure

POST All data and head structure deleted

RETURN null head pointer

1 loop (pList→count not zero)

    1 dltPtr = pList→head

    2 pList→head = dltPtr→link

    3 pList→count = pList→count – 1

    4 release (dltPtr)

No data left in list

2 release(pList)

3 return null pointer

# List ADT

- Bağlı liste implementasyonuna dayalı bir **list ADT** oluşturma
- Bir ADT'nin bir veri tipinden ve verileri manipüle eden operasyonlardan oluştuğunu hatırlayın.
- C'de bir ADT gerçekleştirdiğimizde, uygulama verileri (application data) programcı tarafından kontrol edilir.
  - Bu, uygulama (application) programının veriler için bellek ayırması ve veri düğümünün adresini ADT'ye iletmesini gerektirir.

# List ADT

- C dili "**strongly typed**" olduğu için veri düğüm işaretçisi void pointer olarak iletilmelidir.
- Void pointer kullanmak veriler hakkında hiçbir ayrıntı bilmeden veri düğüm işaretçisini ADT'nin veri yapısında saklamaya olanak tanır.
- Ancak, veri düğümü işaretçisini saklayabilmek tüm sorunlarımızı çözmez.
- Listeler ve diğer birçok veri yapısı, verileri sıralayabilmemizi gerektirir.
- Bir listede, veriler genellikle anahtar sırasıyla saklanır.
- ADT bu sıralamayı yapmak için gerekli bilgiye sahip değil.

# List ADT

- Ayrıca, her veri türü iki anahtarı karşılaştırmak için farklı fonksiyonlar gerektirir.
- Bu sorunun çözümü fonksiyon işaretçilerindedir (**pointers to functions**).
- Uygulama programcısı, veri yapısındaki iki anahtarı karşılaştırmak için bir fonksiyon yazar (Örneğin iki tamsayıyı karşılaştırmak için)
  - ve bu karşılaştırma fonksiyonunun (*compare function*) **adresini** ADT'ye iletir.

# Sample Definition

```
typedef struct node
{
    int  number;
    struct node *link;
}NODE;
```

```
typedef struct
{
    int count;
    NODE *pos;
    NODE *head;
    NODE *rear;
}LIST;
```



# ADT Structure

- ADT daha sonra karşılaştırma fonksiyon adresini liste baş yapısında (**head structure**) **metadata** olarak tutar
  - ve verilerin karşılaştırılması gerektiğinde onu kullanır.

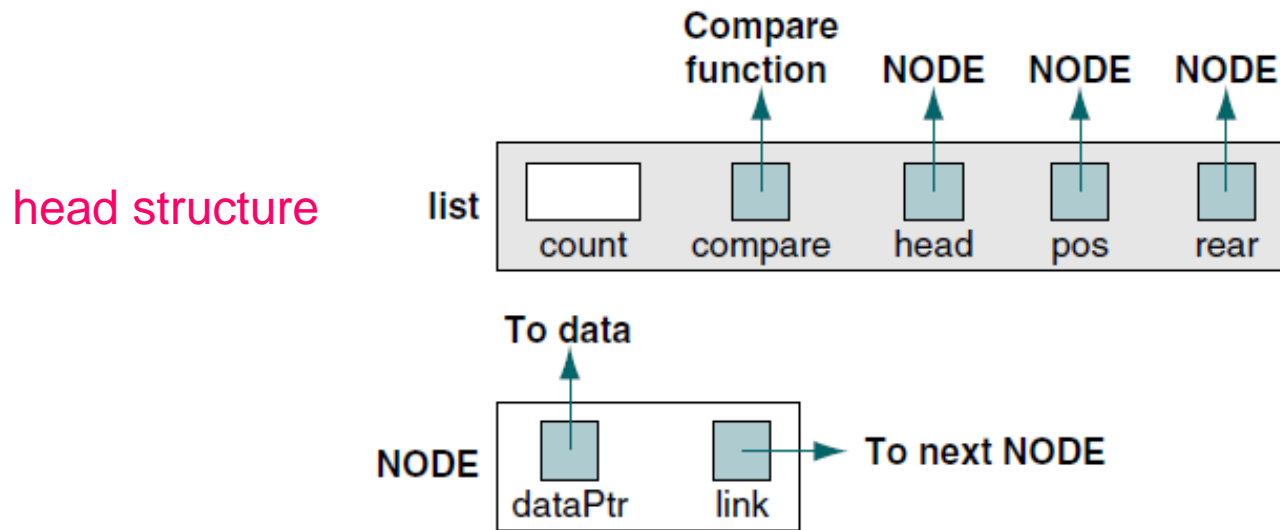
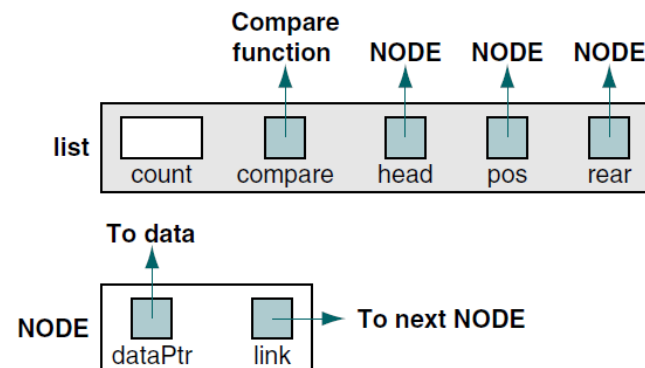


FIGURE 5-15 ADT Structure

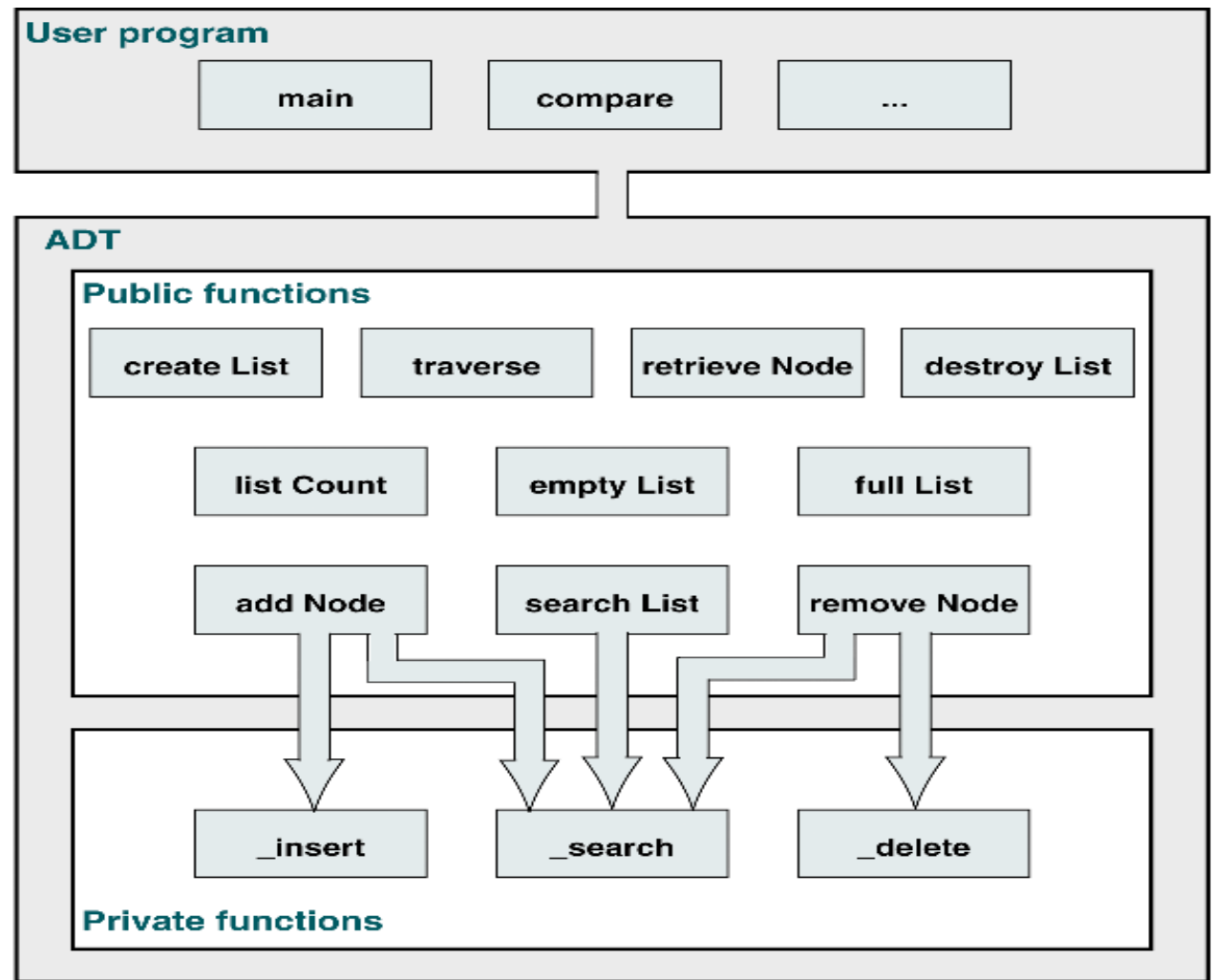
# List ADT Type Definitions

## PROGRAM 5-1 List ADT Type Definitions

```
1 //List ADT Type Definitions
2 typedef struct node
3 {
4     void*      dataPtr;
5     struct node* link;
6 } NODE;
7
8 typedef struct
9 {
10     int    count;
11     NODE*  pos;
12     NODE*  head;
13     NODE*  rear;
14     int    (*compare) (void* argu1, void* argu2);
15 } LIST;
```



# List ADT Functions



Bunlar uygulama kodunda kullanılmayan iç fonksiyonlardır; yalnızca dahili ADT kullanımı içindir.

FIGURE 5-16 List ADT Functions

# List ADT Prototype Declarations

## PROGRAM 5-2 List ADT Prototype Declarations

```
1  //Prototype Declarations
2  LIST* createList    (int (*compare)
3                      (void* argu1, void* argu2));
4  LIST* destroyList   (LIST* list);
5
6  int    addNode       (LIST* pList, void* dataInPtr);
7
8  bool   removeNode    (LIST*  pList,
9                      void*   keyPtr,
10                     void**  dataOutPtr);
11
12  bool   searchList    (LIST*  pList,
13                     void*   pArgu,
14                     void**  pDataOut);
```

*continued*

## PROGRAM 5-2 List ADT Prototype Declarations *(continued)*

```
15
16     bool  retrieveNode (LIST*  pList,
17                          void*  pArgu,
18                          void** dataOutPtr);
19
20     bool  traverse      (LIST*  pList,
21                          int     fromWhere,
22                          void** dataOutPtr);
23
24     int   listCount     (LIST*  pList);
25     bool  emptyList     (LIST*  pList);
26     bool  fullList      (LIST*  pList);
27
28     static int _insert   (LIST* pList,
29                          NODE* pPre,
30                          void* dataInPtr);
31
32     static void _delete  (LIST*  pList,
33                          NODE*  pPre,
34                          NODE*  pLoc,
35                          void** dataOutPtr);
36     static bool _search  (LIST*  pList,
37                          NODE** pPre,
38                          NODE** pLoc,
39                          void*  pArgu);
40 //End of List ADT Definitions
```

# Create List

## PROGRAM 5-3 Create List

- «head structure» için bellek tahsis eder
- İşaretçileri ilklendirir
- Oluşturulan yapının adresini çağıran (calling) fonksiyona döndürür

```
1  /*===== createList =====
2  Allocates dynamic memory for a list head
3  node and returns its address to caller
4  Pre    compare is address of compare function
5         used to compare two nodes.
6  Post   head has allocated or error returned
7  Return head node pointer or null if overflow

8  */
9  LIST* createList
10     (int (*compare) (void* argu1, void* argu2))
11  {
12     //Local Definitions
13     LIST* list;
14
15     //Statements
16     list = (LIST*) malloc (sizeof (LIST));
17     if (list)
18     {
19         list->head    = NULL;
20         list->pos      = NULL;
21         list->rear     = NULL;
22         list->count    = 0;
23         list->compare = compare;
24     } // if
25
26     return list;
27 } // createList
```

# Add Node

- «Düğüm ekle» aslında listeye eklenecek veriyi alan ve ekleme noktasını bulmak için listede arama yapan daha üst düzey (**higher-level**) bir kullanıcı arabirimi (**user-interface**) fonksiyonudur.
- Bu ADT'de, yinelenen anahtarların listeye eklenmesini önleme tercih edilmiştir. (Anahtar değeri eşsiz olmalı)
- Bu, üç olası dönüş değerine yol açar:
  - -1 → dinamik bellek taşma durumunu (overflow)
  - 0 → başarılı eklemeyi (success)
  - +1, yinelenen bir anahtarı belirtir (duplicate key)

Bu dönüş değerlerini doğru şekilde yorumlamak programcının sorumluluğundadır.

# Add Node

## PROGRAM 5-4 Add Node

```
1  /*===== addNode =====
2  Inserts data into list.
3      Pre      pList is pointer to valid list
4              dataInPtr pointer to insertion data
5      Post     data inserted or error
6      Return   -1 if overflow
7              0 if successful
8              1 if dupe key
9  */
10 int addNode (LIST* pList, void* dataInPtr)
11 {
12     //Local Definitions
13     bool found;
14     bool success;
15
16     NODE* pPre;
17     NODE* pLoc;
18
19     //Statements
20     found = _search (pList, &pPre, &pLoc, dataInPtr);
21     if (found)
22         // Duplicate keys not allowed
23         return (+1);
24
25     success = _insert (pList, pPre, dataInPtr);
26     if (!success)
27         // Overflow
28         return (-1);
29     return (0);
30 } // addNode
```



# Internal Insert Function

## PROGRAM 5-5 Internal Insert Function

1	/*===== _insert =====
2	Inserts data pointer into a new node.
3	Pre     pList pointer to a valid list
4	pPre pointer to data's predecessor
5	dataInPtr data pointer to be inserted
6	Post    data have been inserted in sequence
7	Return boolean, true if successful,
8	false if memory overflow

*continued*

## PROGRAM 5-5 Internal Insert Function (*continued*)

```
9  */
10 static bool _insert (LIST* pList, NODE* pPre,
11                     void* dataInPtr)
12 {
13     //Local Definitions
14     NODE* pNew;
15
16     //Statements
17     if (!(pNew = (NODE*) malloc(sizeof(NODE))))
18         return false;
19
20     pNew->dataPtr    = dataInPtr;
21     pNew->link       = NULL;
22
23     if (pPre == NULL)
24     {
25         // Adding before first node or to empty list.
26         pNew->link    = pList->head;
27         pList->head    = pNew;
28         if (pList->count == 0)
29             // Adding to empty list. Set rear
30             pList->rear = pNew;
31     } // if pPre
32     else
33     {
34         // Adding in middle or at end
35         pNew->link    = pPre->link;
36         pPre->link    = pNew;
37
38         // Now check for add at end of list
39         if (pNew->link == NULL)
40             pList->rear = pNew;
41     } // if else
42
43     (pList->count)++;
44     return true;
45 } // _insert
```

# Remove Node

- «Remove node» da aynı zamanda bir üst seviye, kullanıcı arayüzü fonksiyonudur.
- Silme işlemini tamamlamak için liste arama ve düğümü sil alt programlarını çağırır.
- «Remove node» iki olası tamamlama durumu vardır:
  - ya başarılı olur (true)
  - ya da başarısız (false) olur çünkü silinecek veriler bulunamamıştır

# Remove Node

## PROGRAM 5-6 Remove Node

```
1  /*===== removeNode =====
2      Removes data from list.
3      Pre      pList pointer to a valid list
4                keyPtr pointer to key to be deleted
5                dataOutPtr pointer to data pointer
6      Post     Node deleted or error returned.
7      Return   false not found; true deleted
8  */
9  bool removeNode (LIST* pList, void* keyPtr,
10                  void** dataOutPtr)
11  {
12      //Local Definitions
13      bool found;
14
15      NODE* pPre;
16      NODE* pLoc;
17
18      //Statements
19      found = _search (pList, &pPre, &pLoc, keyPtr);
20      if (found)
21          _delete (pList, pPre, pLoc, dataOutPtr);
22
23      return found;
24  } // removeNode
```

# Internal Delete Function

## PROGRAM 5-7 Internal Delete Function

Dahili silme fonksiyonu (\_delete), tanımlanan düğümü dinamik bellekten fiziksel olarak silmek için «Remove Node» tarafından çağrılır.

Veri silindiğinde, ona işaret eden bir pointer, çağırın fonksiyonuna geri döner ve çağrıdaki son parametre tarafından belirtilen değişken konumuna (dataOutPtr) yerleştirilir.

1	/*===== _delete =====
2	Deletes data from a list and returns
3	pointer to data to calling module.
4	Pre     pList pointer to valid list.
5	pPre pointer to predecessor node
6	pLoc pointer to target node
7	dataOutPtr pointer to data pointer
8	Post     Data have been deleted and returned
9	Data memory has been freed
10	*/
11	void _delete (LIST* pList, NODE* pPre,
12	NODE* pLoc, void** dataOutPtr)
13	{
14	//Statements
15	*dataOutPtr = pLoc->dataPtr;
16	if (pPre == NULL)
17	// Deleting first node
18	pList->head = pLoc->link;
19	else
20	// Deleting any other node
21	pPre->link = pLoc->link;
22	
23	// Test for deleting last node
24	if (pLoc->link == NULL)
25	pList->rear = pPre;
26	
27	(pList->count)--;
28	free (pLoc);
29	
30	return;
31	} // _delete

# Search List

- Listedeki belirli bir düğümü bulup adresini ve selefinin adresini, çağıran fonksiyona geri gönderen üst seviye bir kullanıcı arabirimi fonksiyonudur.
- Üç parametreye ihtiyaç duyar:
  - aranacak liste
  - arama argümanı
  - Veri işaretçisini tutacak işaretçinin adresi (pointer to pointer)

# Search List

## PROGRAM 5-8 Search User Interface

```
1  /*===== searchList =====
2  Interface to search function.
3      Pre    pList pointer to initialized list.
4             pArgu pointer to key being sought
5      Post   pDataOut contains pointer to found data
6             -or- NULL if not found
7      Return boolean true successful; false not found
8  */
```

```
9  bool searchList (LIST* pList, void* pArgu,
10                  void** pDataOut)
11  {
12  //Local Definitions
13      bool found;
14
15      NODE* pPre;
16      NODE* pLoc;
17
18  //Statements
19      found = _search (pList, &pPre, &pLoc, pArgu);
20      if (found)
21          *pDataOut = pLoc->dataPtr;
22      else
23          *pDataOut = NULL;
24      return found;
25  } // searchList
```

# Internal Search Function

- Gerçek arama işi yalnızca ADT içinde mevcut olan dahili bir arama fonksiyonu (`_search`) ile yapılır.
- Arama argümanının bir düğümdeki anahtara eşit olup olmadığını belirlemek için kullanıcı programı tarafında oluşturulan bir karşılaştırma fonksiyonunu kullanır (ihtiyaç duyar).



# Internal Search Function

## PROGRAM 5-9 Internal Search Function

```
1  /*===== _search =====
2     Searches list and passes back address of node
3     containing target and its logical predecessor.
4     Pre      pList pointer to initialized list
5              pPre  pointer variable to predecessor
6              pLoc  pointer variable to receive node
7              pArgu pointer to key being sought
8     Post     pLoc points to first equal/greater key
9             -or- null if target > key of last node
10             pPre points to largest node < key
11             -or- null if target < key of first node
12     Return  boolean true found; false not found
13
14  */
15  bool _search (LIST*  pList, NODE** pPre,
16               NODE** pLoc, void*  pArgu)
17  {
18      //Macro Definition
19      #define COMPARE \
20          ( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )
```

*continued*

## PROGRAM 5-9 Internal Search Function (continued)

### Compare fonksiyonu

- $\text{arg} < \text{key} \rightarrow -1$
- $\text{Arg} = \text{key} \rightarrow 0$
- $\text{Arg} > \text{key} \rightarrow +1$

döndürür

```
21
22 #define COMPARE_LAST \
23     ((* pList->compare) (pArgu, pList->rear->dataPtr))
24
25 //Local Definitions
26     int result;
27
28 //Statements
29     *pPre = NULL;
30     *pLoc = pList->head;
31     if (pList->count == 0)
32         return false;
33
34     // Test for argument > last node in list
35     if ( COMPARE_LAST > 0)
36     {
37         *pPre = pList->rear;
38         *pLoc = NULL;
39         return false;
40     } // if
41
42     while ( (result = COMPARE) > 0 )
43     {
44         // Have not found search argument location
45         *pPre = *pLoc;
46         *pLoc = (*pLoc)->link;
47     } // while
48
49     if (result == 0)
50         // argument found--success
51         return true;
52     else
53         return false;
54 } // _search
```

# Retrieve Node

PROGRAM 5-10 Retrieve Node

```
1  /*===== retrieveNode =====
2   This algorithm retrieves data in the list without
3   changing the list contents.
4   Pre    pList pointer to initialized list.
5          pArgu pointer to key to be retrieved
6   Post   Data (pointer) passed back to caller
7   Return boolean true success; false underflow
8  */
9  static bool retrieveNode (LIST*  pList,
10                           void*  pArgu,
11                           void** dataOutPtr)
12  {
13  //Local Definitions
14    bool  found;
15
16    NODE* pPre;
17    NODE* pLoc;
18
19  //Statements
20    found = _search (pList, &pPre, &pLoc, pArgu);
21    if (found)
22    {
23        *dataOutPtr = pLoc->dataPtr;
24        return true;
25    } // if
26
27    *dataOutPtr = NULL;
28    return false;
29  } // retrieveNode
```

# Status functions

- Uygulama programcısı liste yapısına erişmediğinden, listenin durumunu tespit etmek için kullanılabilecek üç durum fonksiyonu sağlanır.
  - Empty List
  - Full List
  - List Count

# Empty List

## PROGRAM 5-11 Empty List

```
1  /*===== emptyList =====
2     Returns boolean indicating whether or not the
3     list is empty
4     Pre    pList is a pointer to a valid list
5     Return boolean true empty; false list has data
6  */
7  bool emptyList (LIST* pList)
8  {
9      //Statements
10     return (pList->count == 0);
11 } // emptyList
```

# Full List

## PROGRAM 5-12 Full List

C, dinamik bellekteki boş alanı belirleme olanağı sağlamadığından, yalnızca bir düğüm için yer ayırmayı deneriz; işe yararsa, en az bir düğüm için daha yer olduğunu anlamış oluruz.

```
1  /*===== fullList =====
2  Returns boolean indicating no room for more data.
3  This list is full if memory cannot be allocated for
4  another node.
5      Pre    pList pointer to valid list
6      Return boolean true if full
7              false if room for node
8  */
9  bool fullList (LIST* pList)
10 {
11     //Local Definitions
12     NODE* temp;
13
14     //Statements
15     if ((temp = (NODE*)malloc(sizeof(*(pList->head)))))
16     {
17         free (temp);
18         return false;
19     } // if
20
21     // Dynamic memory full
22     return true;
23
24 } // fullList
```

# List Count

## PROGRAM 5-13 List Count

```
1  /*===== listCount =====  
2  Returns number of nodes in list.  
3  Pre    pList is a pointer to a valid list  
4  Return count for number of nodes in list  
5  */  
6  int listCount(LIST* pList)  
7  {  
8  //Statements  
9  
10     return pList->count;  
11  
12 } // listCount
```

# Traverse

- Programcının liste yapısına erişimi olmadığı için listeyi dolaşmak/gezinmek için bir fonksiyon sağlamamız gerekir.

## PROGRAM 5-14 Traverse List

1	/*===== traverse =====		
2	Traverses a list. Each call either starts at the		
3	beginning of list or returns the location of the		
4	next element in the list.		
5	Pre	pList	pointer to a valid list
6		fromWhere	0 to start at first element
7		dataPtrOut	address of pointer to data
8	Post	if more data, address of next node	

*continued*



# Traverse

## PROGRAM 5-14 Traverse List (continued)

«Traverse»

fonksiyonunun gezinme daha yeni mi başladı yoksa bir gezinmenin ortasında mı olduğunu bilmesi gerekir.

Bu durum fromWhere bayrağıyla kontrol edilir.

Fonksiyon her çağrıldığında, döndürülen geçerli düğümün adresini baş (head) düğümdeki konum (pos) işaretçisinde saklar.

Sonra bir dahaki sefere, listenin başından başlamazsak, bir sonraki düğümü bulmak için konum işaretçisini kullanabiliriz.

```
9      Return true node located; false if end of list
10     */
11     bool traverse (LIST*  pList,
12                   int     fromWhere,
13                   void**  dataPtrOut)
14     {
15         //Statements
16         if (pList->count == 0)
17             return false;
18
19         if (fromWhere == 0)
20         {
21             // Start from first node
22             pList->pos = pList->head;
23             *dataPtrOut = pList->pos->dataPtr;
24             return true;
25         } // if fromwhere
26     else
27     {
28         // Start from current position
29         if (pList->pos->link == NULL)
30             return false;
31         else
32         {
33             pList->pos = pList->pos->link;
34             *dataPtrOut = pList->pos->dataPtr;
35             return true;
36         } // if else
37     } // if fromwhere else
38 } // traverse
```

# Destroy List

## PROGRAM 5-15 Destroy List

```
1  /*===== destroyList =====
2   Deletes all data in list and recycles memory
3   Pre    List is a pointer to a valid list.
4   Post   All data and head structure deleted
5   Return null head pointer
6  */
7  LIST* destroyList (LIST* pList)
8  {
9   //Local Definitions
10   NODE* deletePtr;
11
12  //Statements
13   if (pList)
14   {
15       while (pList->count > 0)
16       {
17           // First delete data
18           free (pList->head->dataPtr);
19
20           // Now delete node
21           deletePtr = pList->head;
22           pList->head = pList->head->link;
23           pList->count--;
24           free (deletePtr);
25       } // while
26       free (pList);
27   } // if
28   return NULL;
29 } // destroyList
```

