Linked List

- 각각의 Element가 다음 Element의 위치를 저장하여 이어진 데이터 스트럭쳐
- 연속으로 쭉 이어졌다면 왜 배열(Array)를 안쓸까??
  - Array의 장점? 단점?
- Linked List의 장점? 단점?

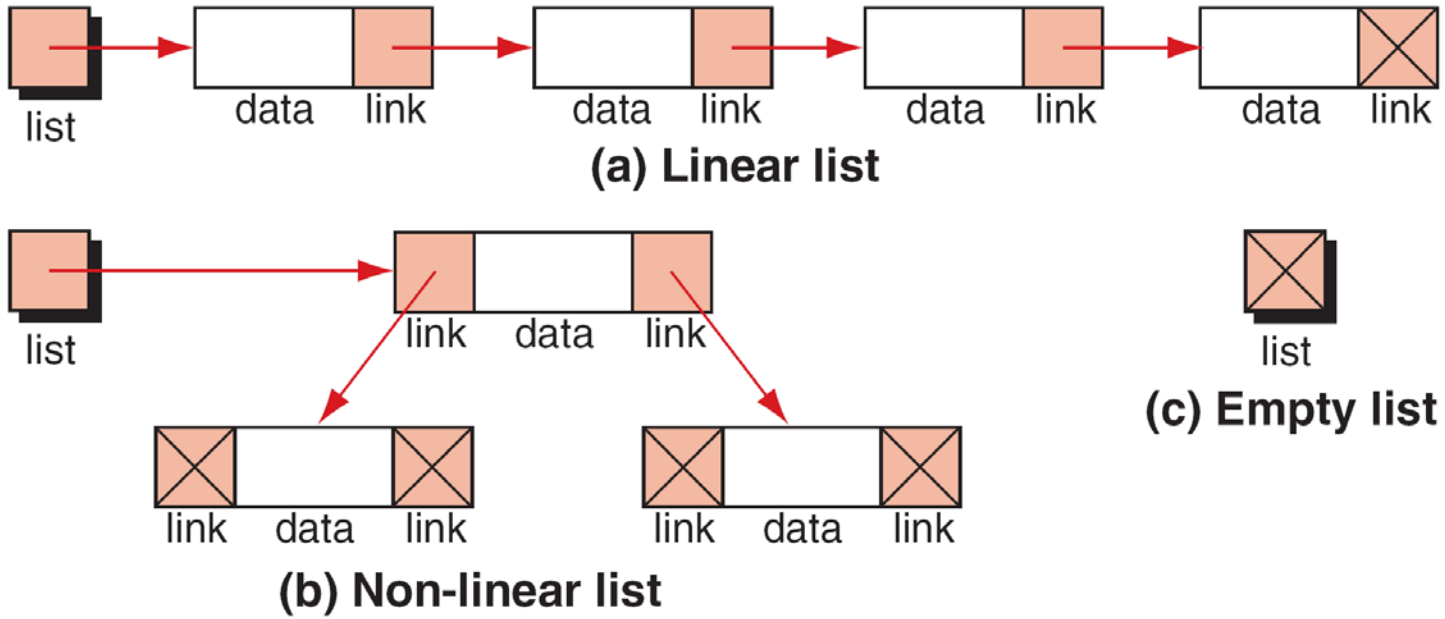
	List	Linked List
Space Complexity	O(n)	
At	O(1)	O(n)
Insert	O(n)	O(1)
Remove	O(n)	O(1)

Array – List Pros and Cons

항목	연결 리스트	배열
임의의 위치 접근	어려움	한번에 가능
임의의 위치 삽입/삭제	쉬움	어려움
가변 개수 데이터 지원	쉬움	최대 수용 가능 개수 이하로만 쉬움
데이터 정렬	어려움	쉬움
기억 공간 차지	(데이터의 크기 + 포인터의 크기 + α) * 연결 개수	최대 수용 가능 개수 * (데이터 당 크기 + β)
사용하지 않는 기억 공간으로 인한 낭비	없음	최대 수용 가능 개수 – 실제로 사용하는 데이터 수
초기화 후 유지/관리	포인터 사용으로 인해 까다로움	배열의 인덱스 값만 조절하면 되므로 쉬움
특정 값을 가지는 데이터 찾기	정렬 여부에 관계없이 어려움	정렬된 배열에서는 쉬움

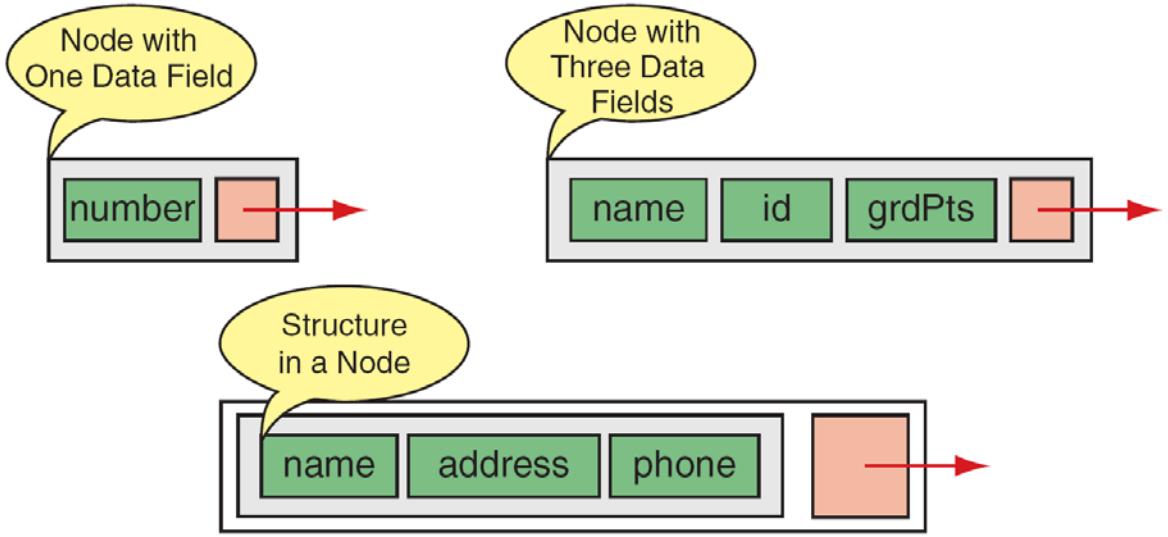
Linked List

- 한개만 가리킬 수도 있고, 여러개를 가리킬 수도 있고…



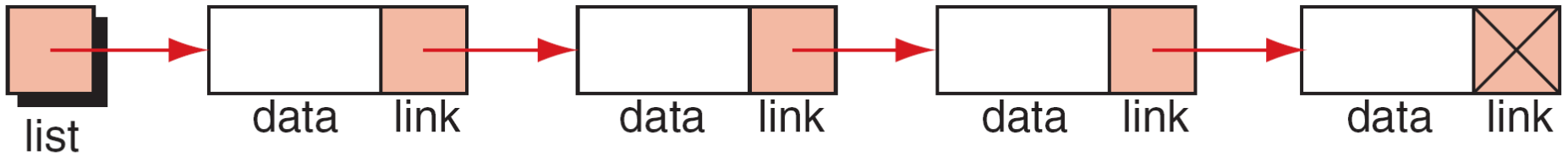
Linked List Node Structures

- 데이터 부분 + 포인터 부분(다른 Element를 가리킴)



General Linear Lists

- 가장 일반적인 직렬 리스트
  - Retrieve
  - Insert
  - Change
  - Delete
  - Traversing
  - Building



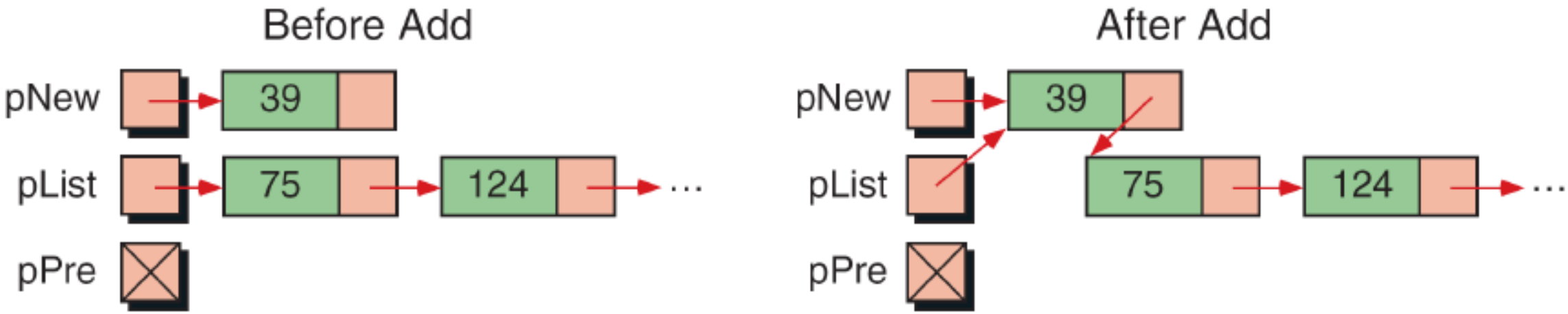
(a) Linear list

Insert a Node to Empty List

```
pNew->link  =  pList ;
pList       =  pNew ;
```

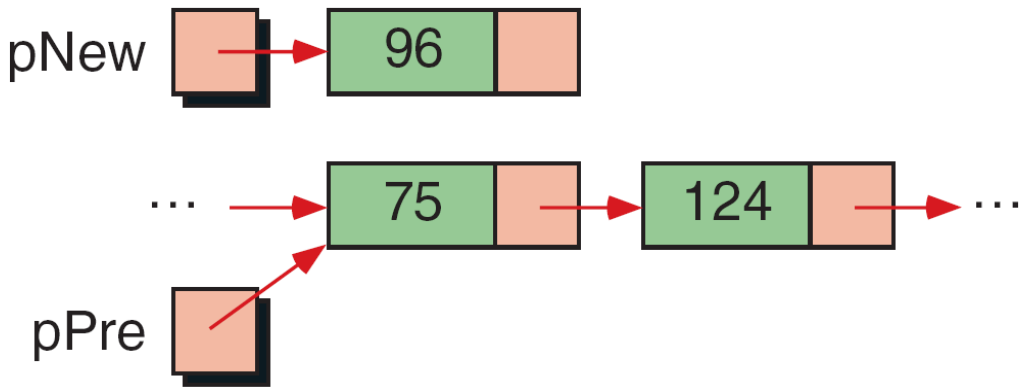
Insert a Node at Beginning

```
pNew->link  =  pList ;
pList       =  pNew;
```

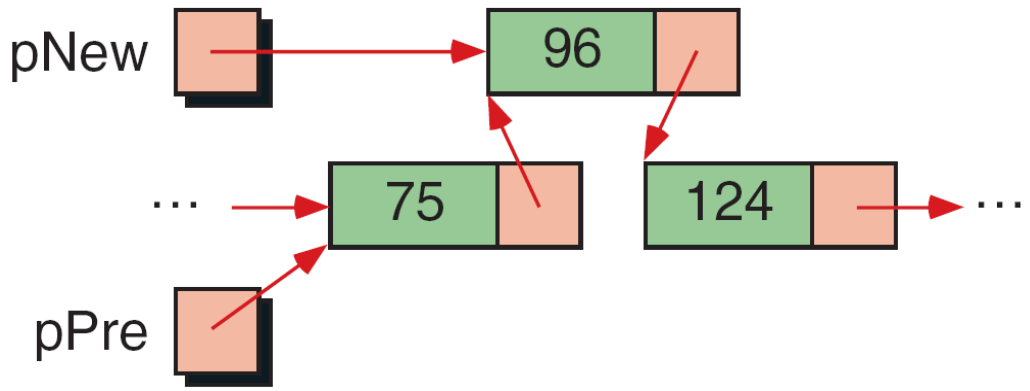


Insert a Node in Middle

```
pNew->link = pPre–>link;
pPre->link = pNew;
```



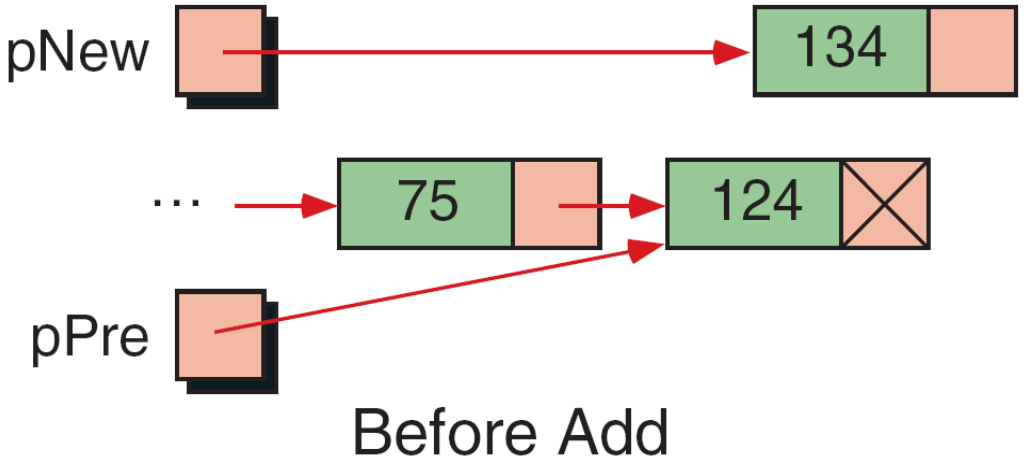
Before Add

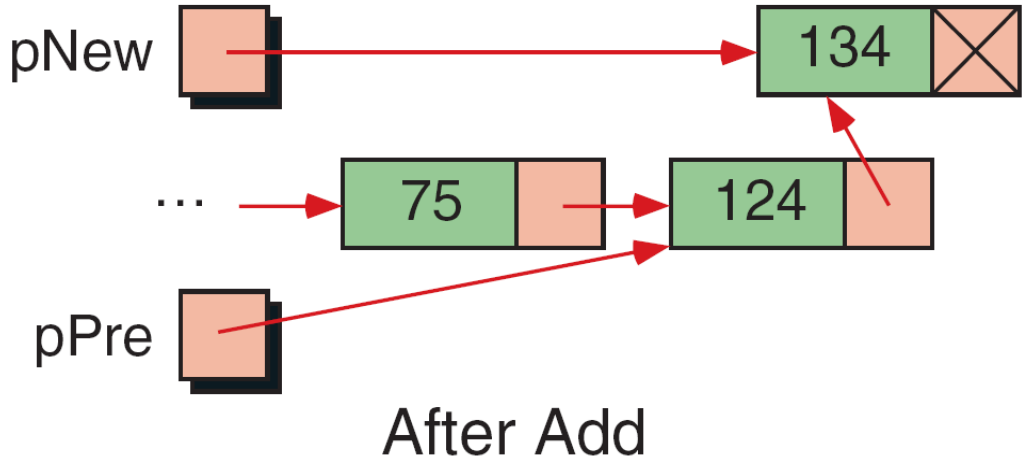


After Add

Insert a Node at End

```
pNew->link = pPre->link;
pPre->link = pNew;
```

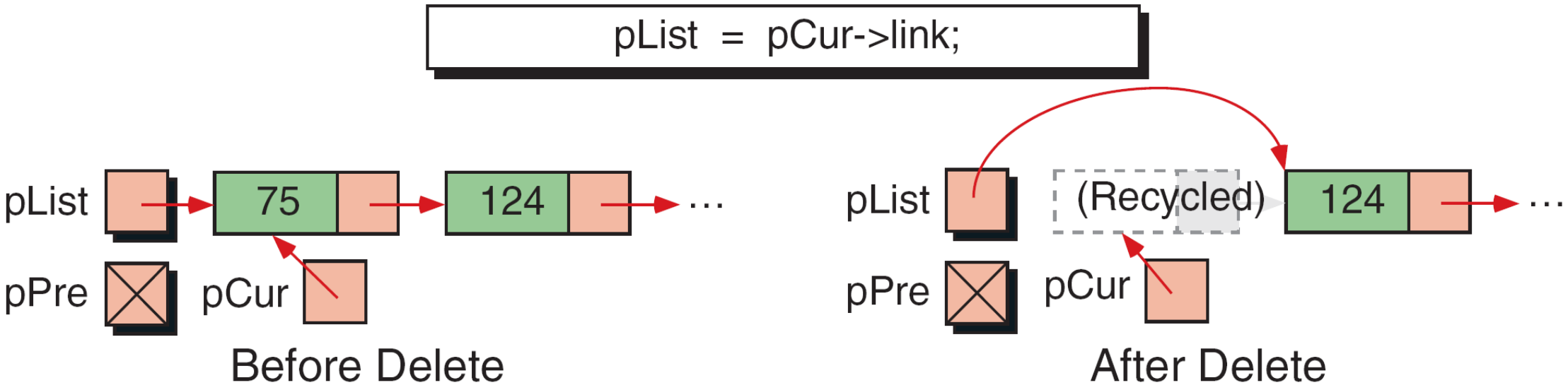




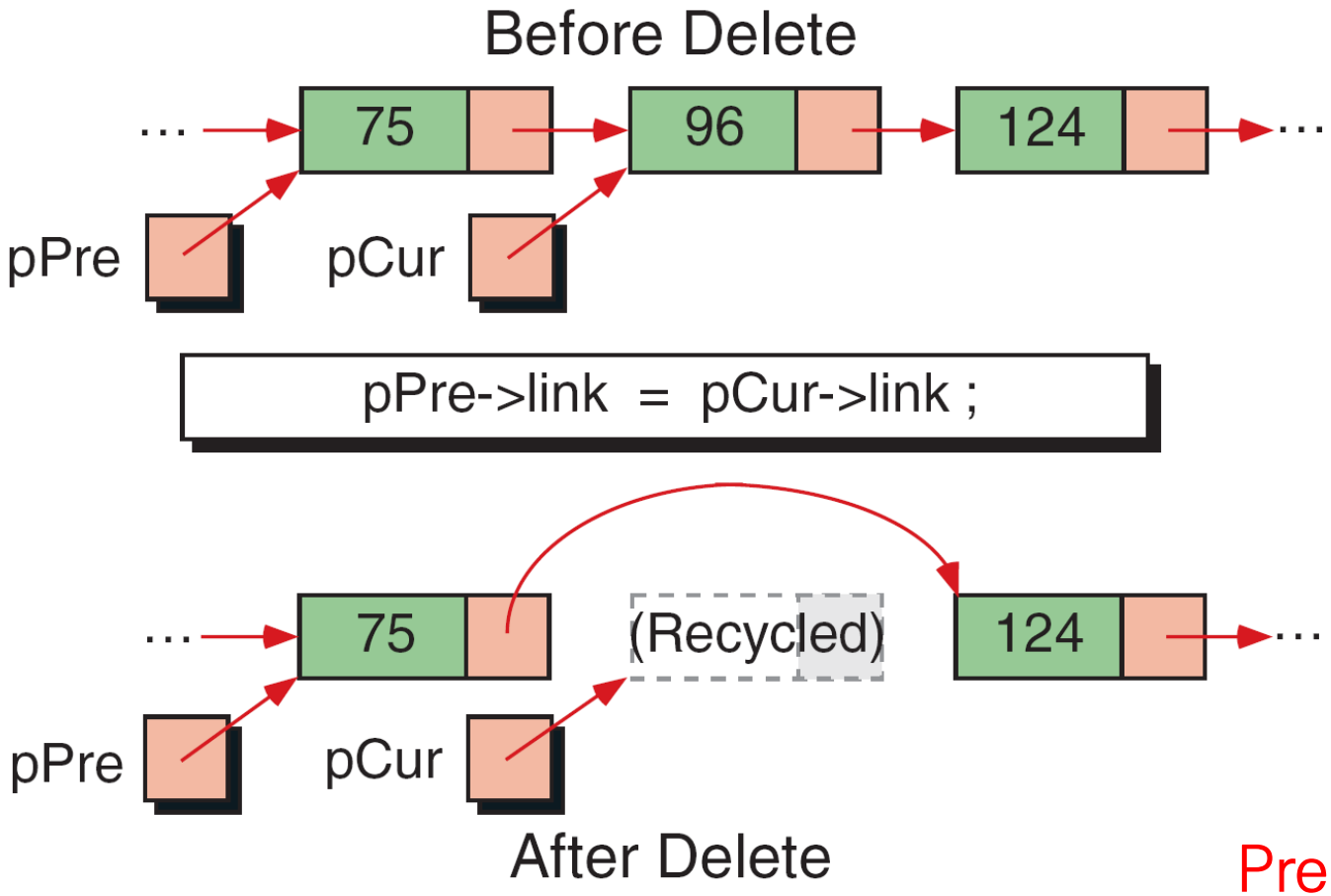
Insert a Node – Code

```
 8  NODE* insertNode (NODE* pList, NODE* pPre, DATA item)
 9  {
10  // Local Declarations
11     NODE* pNew;
12
13  // Statements
14     if (!(pNew  = (NODE*)malloc(sizeof(NODE))))
15         printf("\aMemory overflow in insert\n"),
16                exit (100);
17
18     pNew->data = item;
19     if (pPre == NULL)
20        {
21         // Inserting before first node or to empty list
22         pNew->link  = pList;
23         pList       = pNew;
24        } // if pPre
25     else
26        {
27         // Inserting in middle or at end
28         pNew->link = pPre->link;
29         pPre->link = pNew;
30        } // else
31     return pList;
32  }  // insertNode
```

Delete First Node



Delete – General Case



Pre, Cur 두 개가
같이 움직이는 거 주의!!

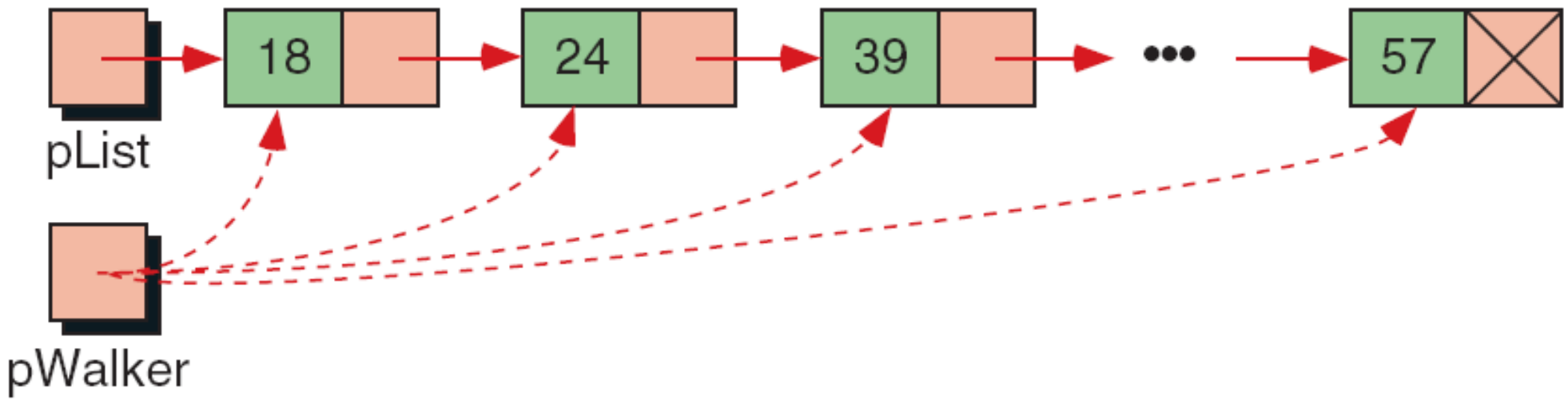
Delete a Node – Code

```
1  /* ==================== deleteNode ====================
2     This function deletes a single node from the link list.
3        Pre   pList is a pointer to the head of the list
4              pPre points to node before the delete node
5              pCur points to the node to be deleted
6        Post  deletes and recycles pCur
7              returns the head pointer
8  */
9  NODE* deleteNode (NODE* pList, NODE* pPre, NODE* pCur)
10 {
11 // Statements
12    if (pPre == NULL)
13       // Deleting first node
14       pList = pCur->link;
15    else
16       // Deleting other nodes
17       pPre->link = pCur->link;
18    free  (pCur);
19    return pList;
20 }  // deleteNode
```

Search Linear List

```
//리스트 pList가 있다.
struct Node *pWalker = pList;
while(pWalker)
{
        If(pWalker->data == 찾는 데이터)
                return 1;
        pWalker = pWalker->next;
}
return 0;
```

Linear List Traversal



Print Linear List

```
5  void printList (NODE* pList)
6  {
7  // Local Declarations
8     NODE* pWalker;
9
10 // Statements
11    pWalker = pList;
12    printf("List contains:\n");
13
14    while (pWalker)
15       {
16        printf("%3d ", pWalker->data.key);
17        pWalker = pWalker->link;
18       } // while
19    printf("\n");
20    return;
```

Average Linear List

```
 5  double averageList (NODE* pList)
 6  {
 7  // Local Declarations
 8     NODE* pWalker;
 9     int   total;
10     int   count;
11
12  // Statements
13     total   = count = 0;
14     pWalker = pList;
15     while (pWalker)
16        {
17         total += pWalker->data.key;
18         count++;
19         pWalker = pWalker->link;
20        } // while
21     return (double)total / count;
22  } // averageList
```