

Chapter 5

General Linear Lists

Objectives

Upon completion you will be able to:

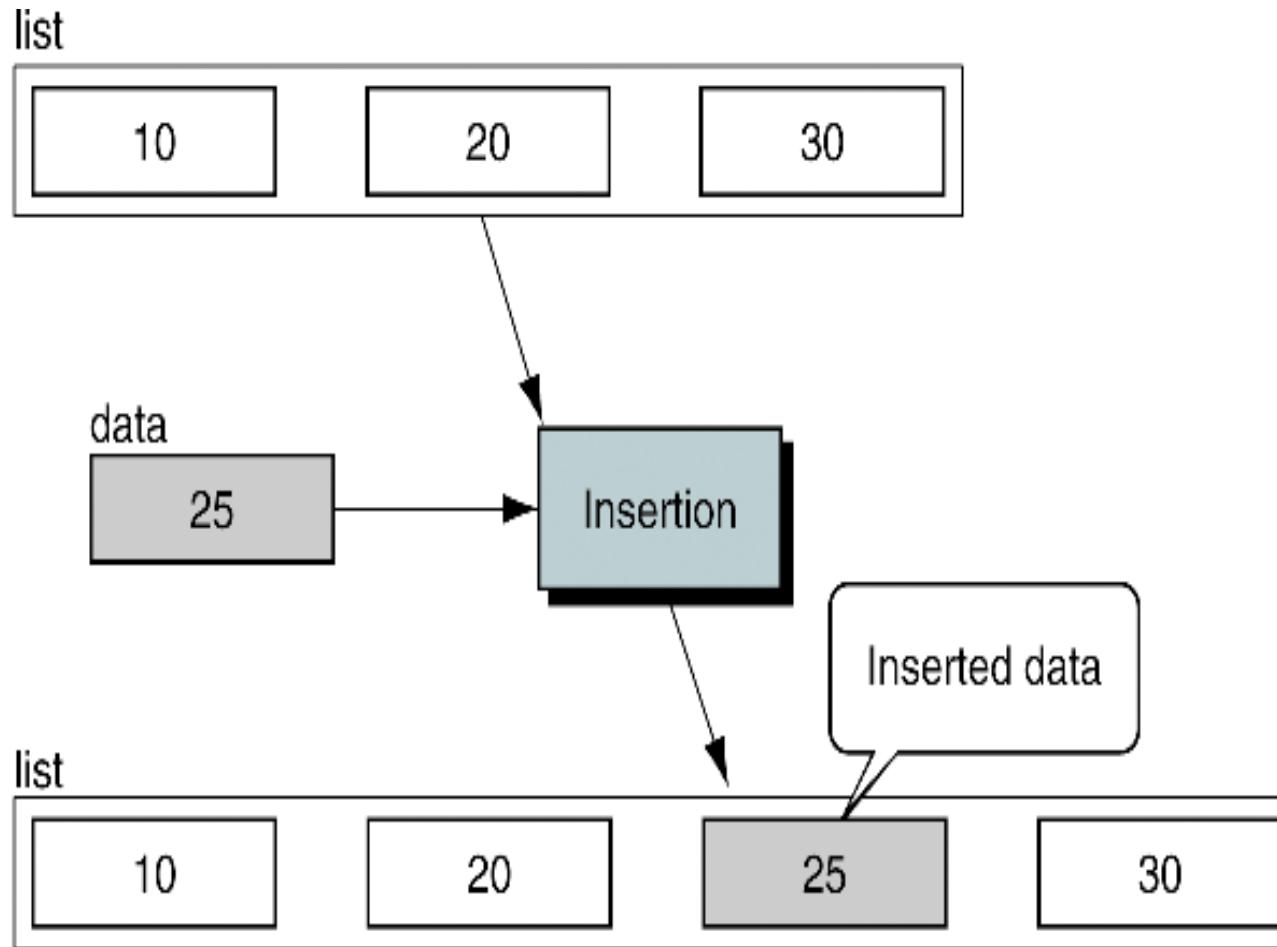
- Explain the design, use, and operation of a linear list
- Implement a linear list using a linked list structure
- Understand the operation of the linear list ADT
- Write application programs using the linear list ADT
- Design and implement different link-list structures

5-1 Basic Operations

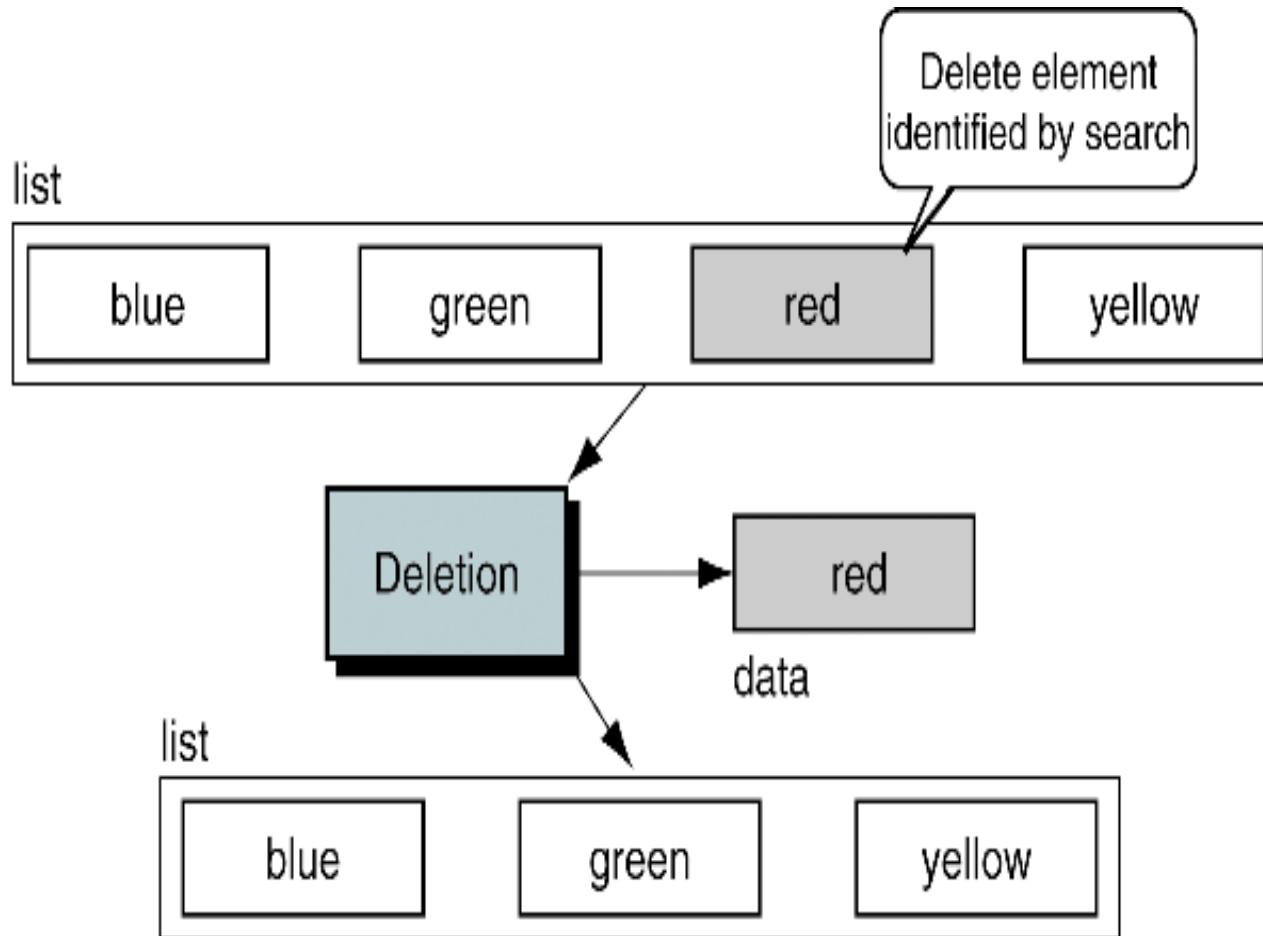
*We begin with a discussion of the basic list operations.
Each operation is developed using before and after
figures to show the changes.*

- Insertion
- Deletion
- Retrieval
- Traversal

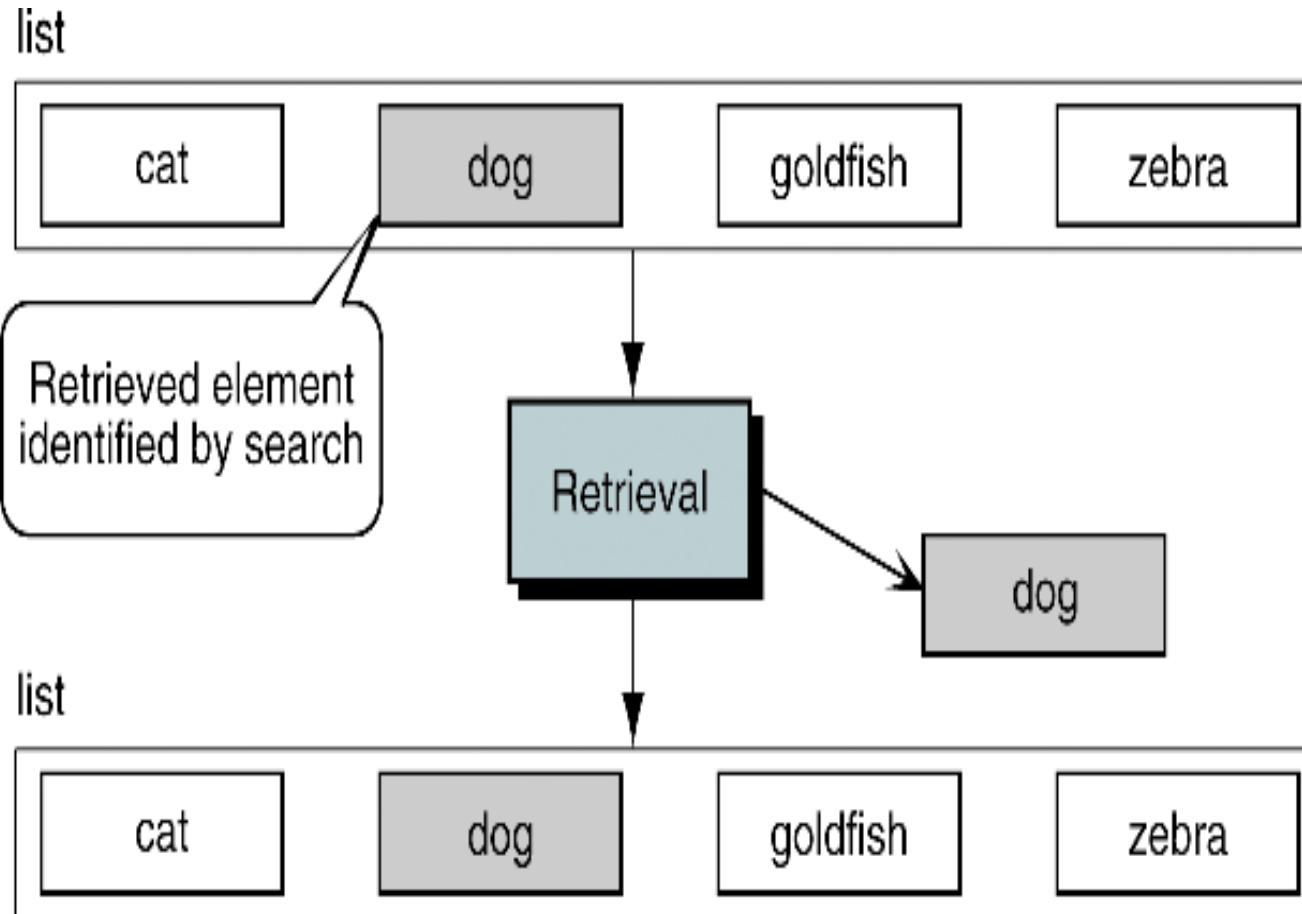
Insertion



Deletion



Retrieval



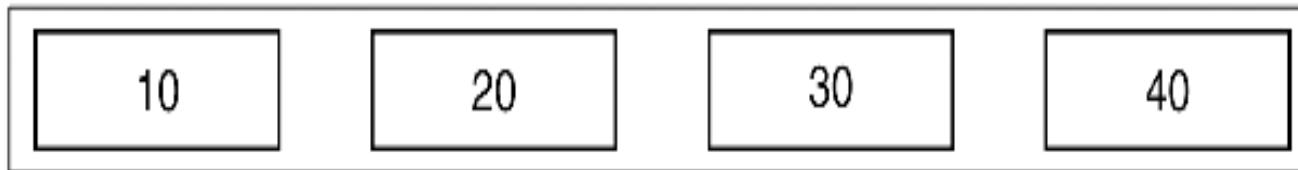
5-2 Implementation

We begin with a discussion of the data structure required to support a list. We then develop the 10 basic algorithms required to build and use a list. In developing the insertion and deletion algorithms, we use extensive examples to demonstrate the analytical steps used in developing algorithms.

- Data Structure
- Algorithms

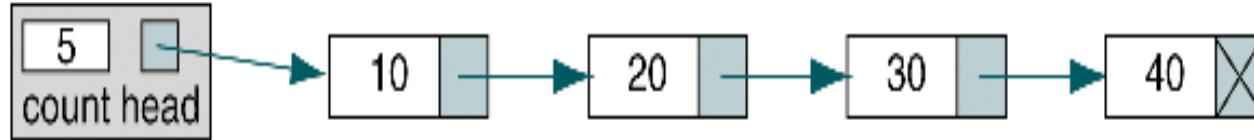
Linked List Implementation of List

List



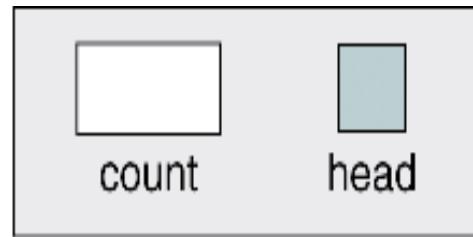
(a) Conceptual view of a list

List

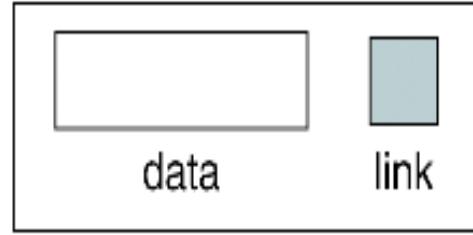


(b) Linked list implementation

Head Node and Data Node



(a) Head structure



(b) Data node structure

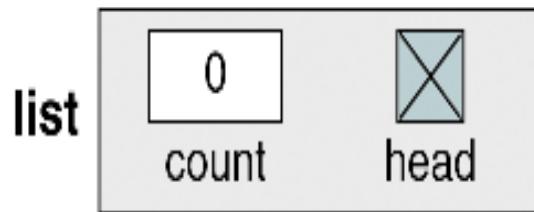
A diagram showing a rectangular box with a folded top-right corner. Inside, there are two parts. The top part is labeled "list" and contains the pseudocode: "count", "head", and "end list". The bottom part is labeled "node" and contains the pseudocode: "data", "link", and "end node".

```
list
    count
    head
end list

node
    data
    link
end node
```

Create List

```
allocate (list)  
set list head to null  
set list count to 0
```

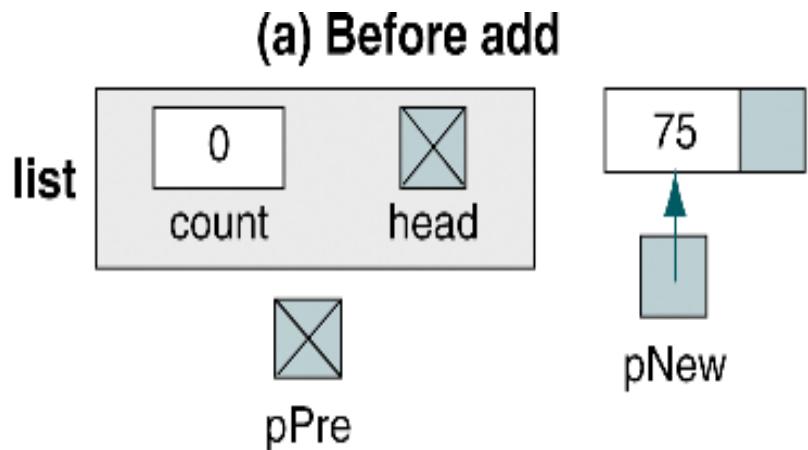


```
Algorithm createList (list)
Initializes metadata for list.

Pre    list is metadata structure passed by reference
Post   metadata initialized

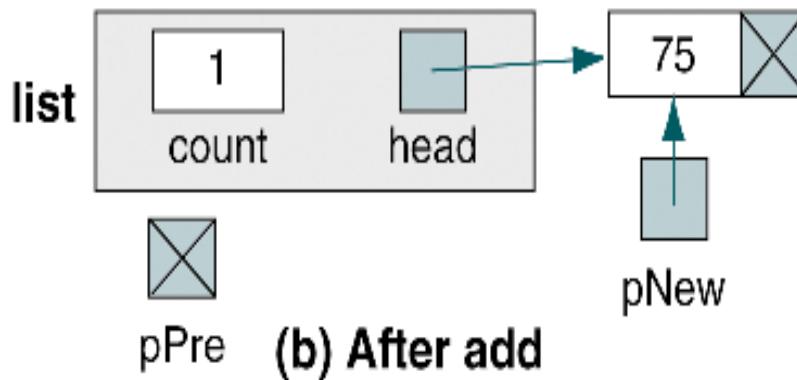
1 allocate (list)
2 set list head to null
3 set list count to 0
end createList
```

Add Node to Empty List

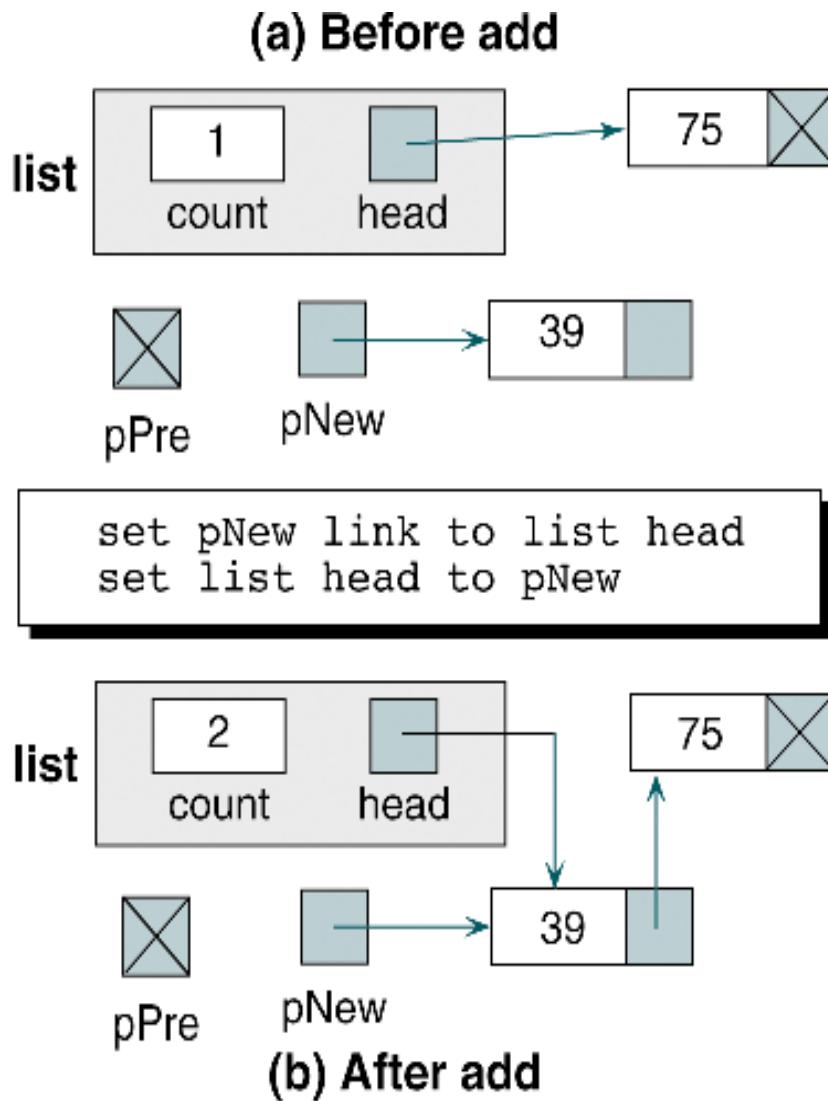


```
set pNew link to list head  
set list head to pNew
```

不是直接設成NULL喔

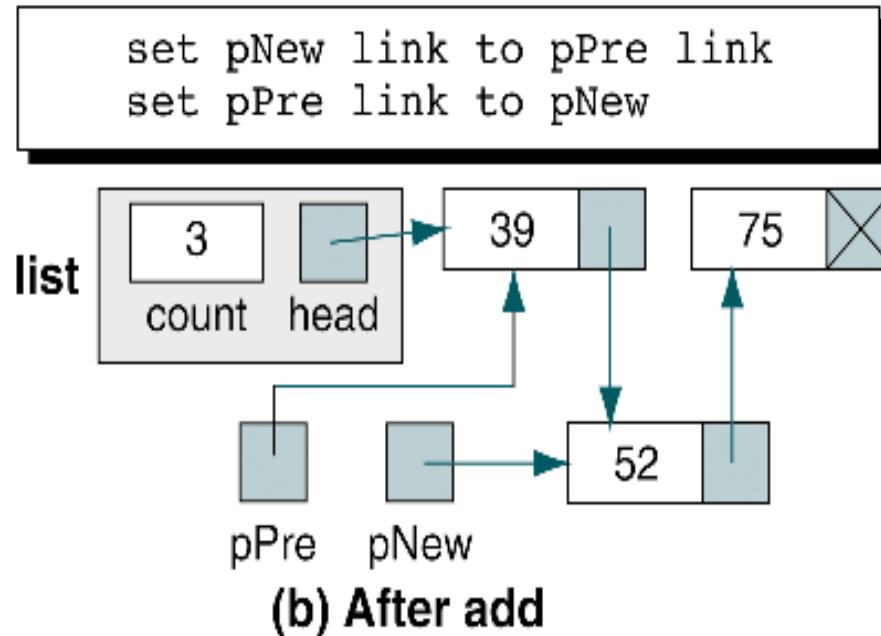
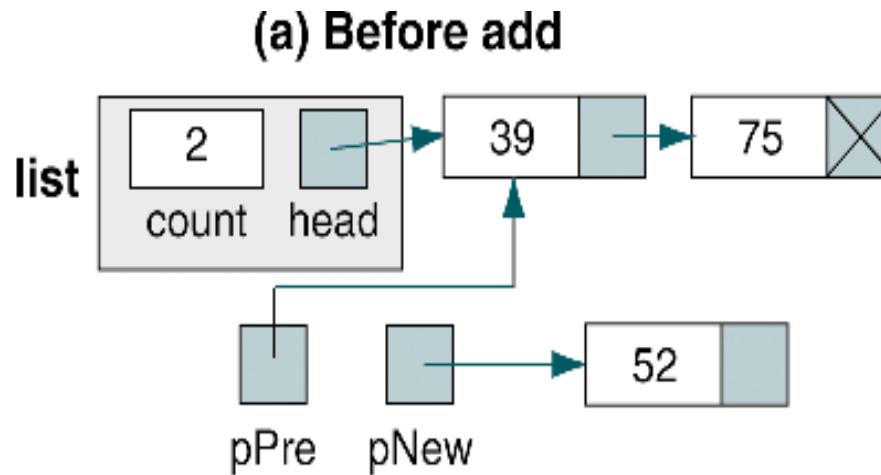


Add Node at Beginning



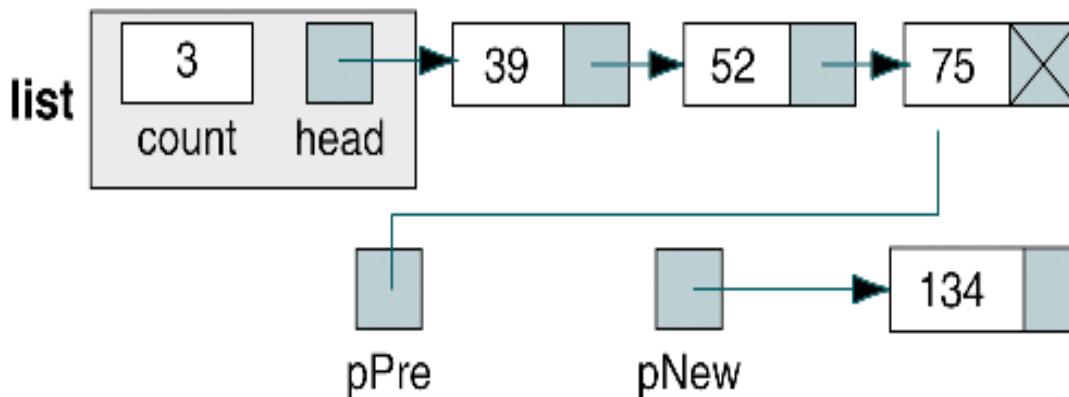
Same as add node
to empty list

Add Node in Middle



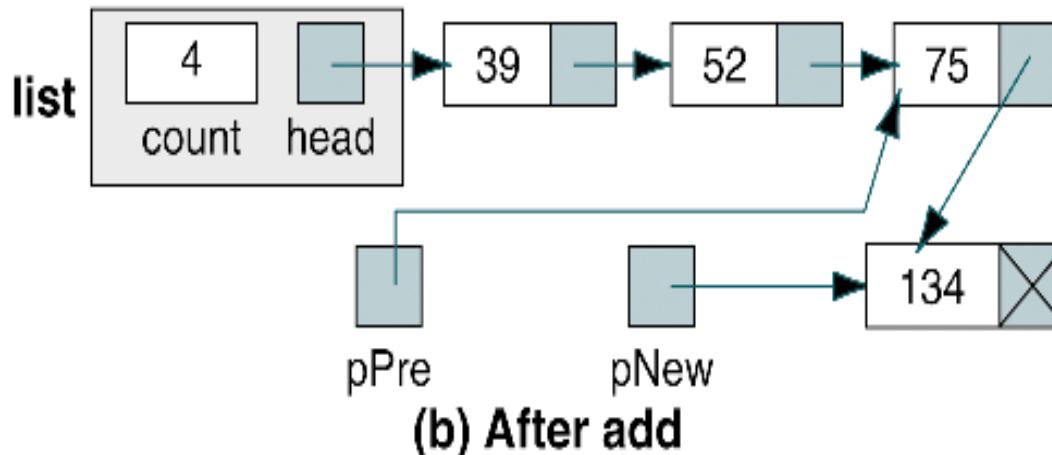
Add Node at End

(a) Before add



```
set pNew link to pPre link  
set pPre link to pNew
```

Same as add node
in middle



Add node

- 1. to empty list
- 2. at begining \Rightarrow $pPre == \text{NULL}$

- 3. in middle
- 4. at end \Rightarrow $pPre != \text{NULL}$

$pPre$ will be set by a search function

Insert List Node

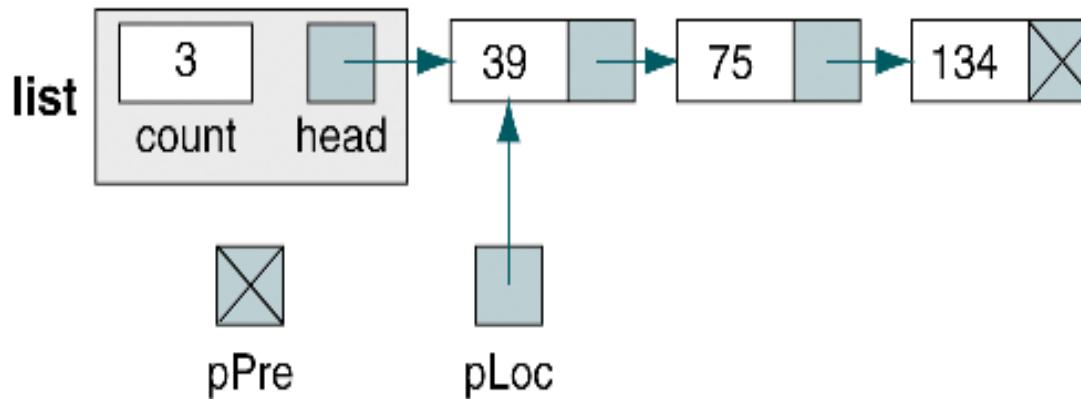
```
Algorithm insertNode (list, pPre, dataIn)
Inserts data into a new node in the list.

    Pre    list is metadata structure to a valid list
           pPre is pointer to data's logical predecessor
           dataIn contains data to be inserted

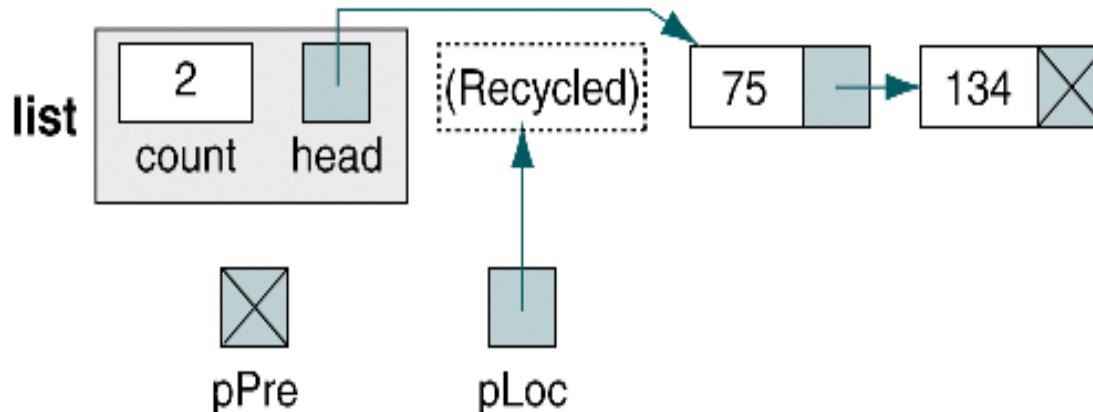
    Post   data have been inserted in sequence
           Return true if successful, false if memory overflow

1 allocate (pNew)
2 set pNew data to dataIn by a search function
3 if (pPre null)
    Adding before first node or to empty list.
    1 set pNew link to list head
    2 set list head to pNew
4 else
    Adding in middle or at end.
    1 set pNew link to pPre link
    2 set pPre link to pNew
5 end if
6 return true
end insertNode
```

Delete First Node

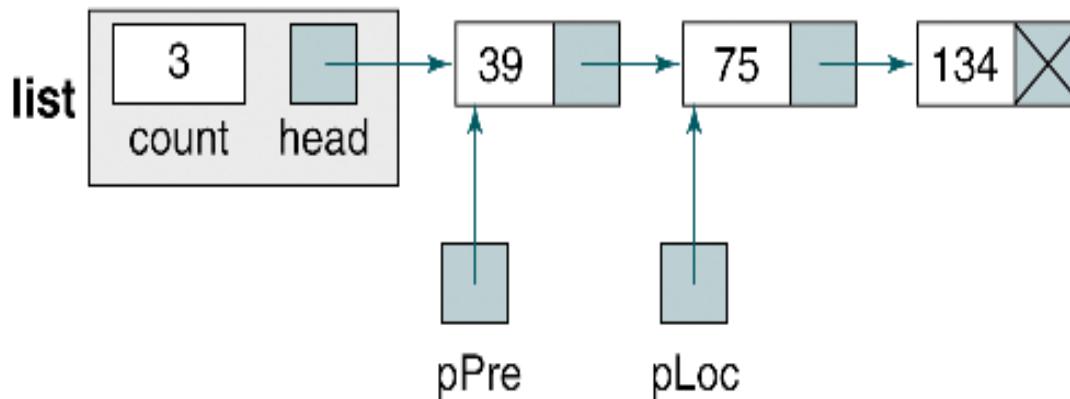


```
set list head to pLoc link
```

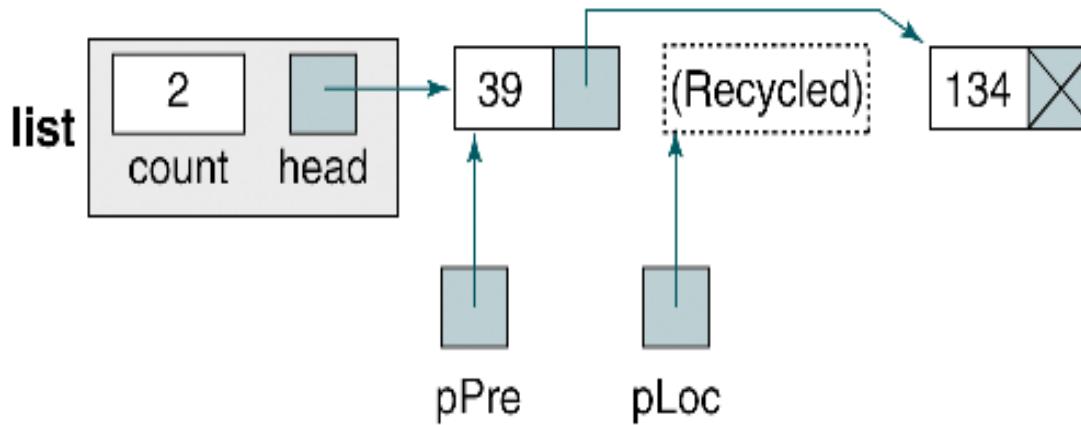


(b) After delete

Delete General Case



set pPre link to pLoc link



(b) After delete

List Delete Node Algorithm

```
Algorithm deleteNode (list, pPre, pLoc, dataOut)
    Deletes data from list & returns it to calling module.

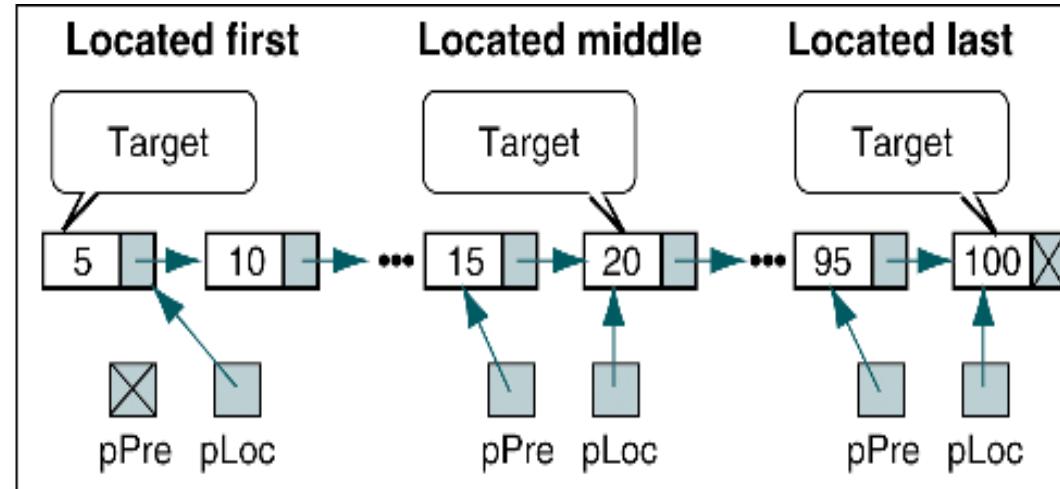
    Pre  list is metadata structure to a valid list
        Pre is a pointer to predecessor node
        pLoc is a pointer to node to be deleted
        dataOut is variable to receive deleted data
    Post data have been deleted and returned to caller

1 move pLoc data to dataOut
2 if (pPre null)
    Deleting first node
    1 set list head to pLoc link
3 else
    Deleting other nodes
    1 set pPre link to pLoc link
4 end if
5 recycle (pLoc)
end deleteNode
```

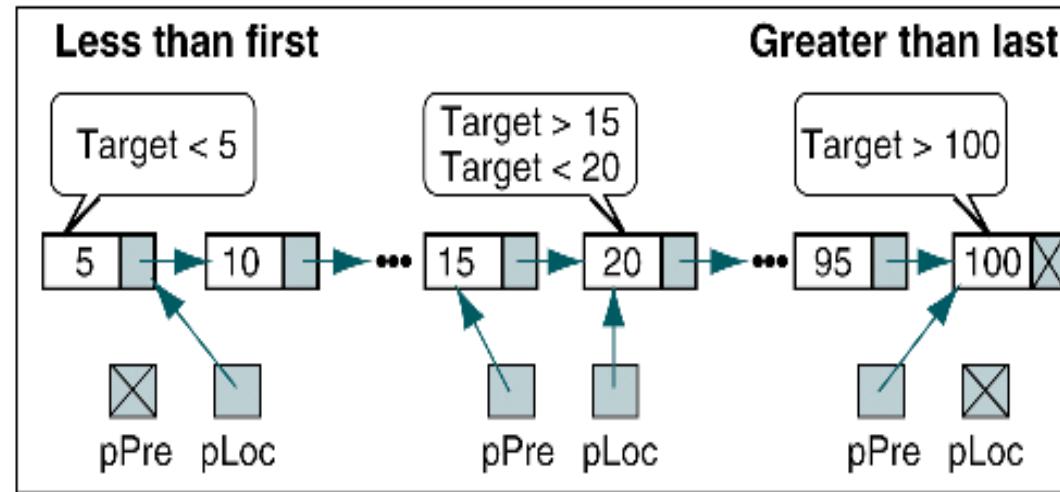
Ordered List Search

這個search function是通用的
(insert跟delete的功能都靠它)

重點是找到pPre跟pLoc的位置



(a) Successful searches (return true) **for delete**



(b) Unsuccessful searches (return false) **for insert**

List Search Results

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

Search List Algorithm

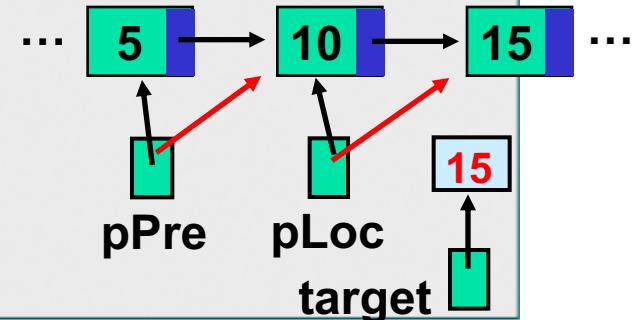
```
Algorithm searchList (list, pPre, pLoc, target)
    Searches list and passes back address of node containing
    target and its logical predecessor.

    Pre    list is metadata structure to a valid list
           pPre is pointer variable for predecessor
           pLoc is pointer variable for current node
           target is the key being sought

    Post   pLoc points to first node with equal/greater 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

           Return true if found, false if not found

1 set pPre to null
2 set pLoc to list head
3 loop (pLoc not null AND target > pLoc key)
    1 set pPre to pLoc
    2 set pLoc to pLoc link
4 end loop
5 if (pLoc null)
    Set return value
    1 set found to false
```



Search List Algorithm (cont.)

```
6 else
    1 if (target equal pLoc key)
        1 set found to true
    2 else
        1 set found to false
    3 end if
7 end if
8 return found
end searchList
```

Retrieve List Node Algorithm

```
Algorithm retrieveNode (list, key, dataOut)
Retrieves data from a list.

Pre    list is metadata structure to a valid list
       key is target of data to be retrieved
       dataOut is variable to receive retrieved data
Post   data placed in dataOut
       -or- error returned if not found
Return true if successful, false if data not found

1 set found to searchList (list, pPre, pLoc, key)
2 if (found)
   1 move pLoc data to dataOut
3 end if
4 return found
end retrieveNode
```

Empty List Algorithm

```
Algorithm emptyList (list)
```

Returns Boolean indicating whether the list is empty.

Pre list is metadata structure to a valid list

Return true if list empty, false if list contains data

```
1 if (list count equal 0)
```

```
    1 return true
```

```
2 else
```

```
    1 return false
```

```
end emptyList
```

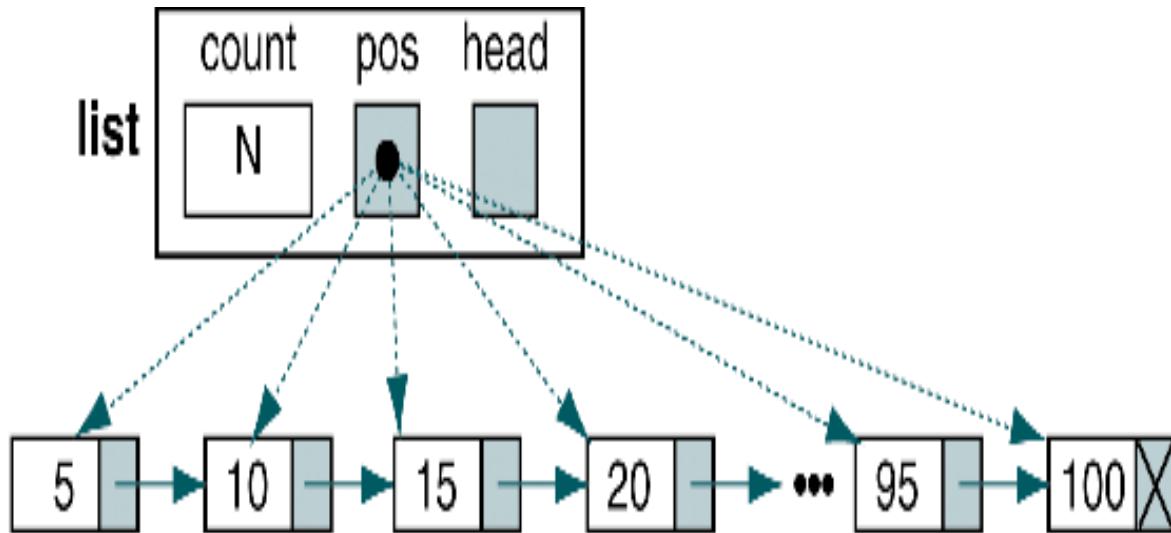
Full List Algorithm

```
Algorithm fullList (list)
Returns Boolean indicating whether or not the list is full.
    Pre    list is metadata structure to a valid list
    Return false if room for new node; true if memory full
1 if (memory full)
    1 return true
2 else
    2 return false
3 end if
4 return true
end fullList
```

List Count Algorithm

```
Algorithm listCount (list)
Returns integer representing number of nodes in list.
    Pre    list is metadata structure to a valid list
    Return count for number of nodes in list
1 return (list count)
end listCount
```

Traverse List



```
Algorithm getNext (list, fromWhere, dataOut)
Traverses a list. Each call returns the location of an
element in the list.
```

Pre list is metadata structure to a valid list
 fromWhere is 0 to start at the first element
 dataOut is reference to data variable

Post dataOut contains data and true returned
 -or- if end of list, returns false

Traverse List (cont.)

```
Return true if next element located  
false if end of list  
1 if (empty list)  
1 return false  
2 if (fromWhere is beginning)  
Start from first  
1 set list pos to list head  
2 move current list data to dataOut  
3 return true  
3 else  
Continue from pos  
1 if (end of list)  
End of List  
1 return false  
2 else  
1 set list pos to next node  
2 move current list data to dataOut  
3 return true  
3 end if  
4 end if  
end getNext
```

Destroy List

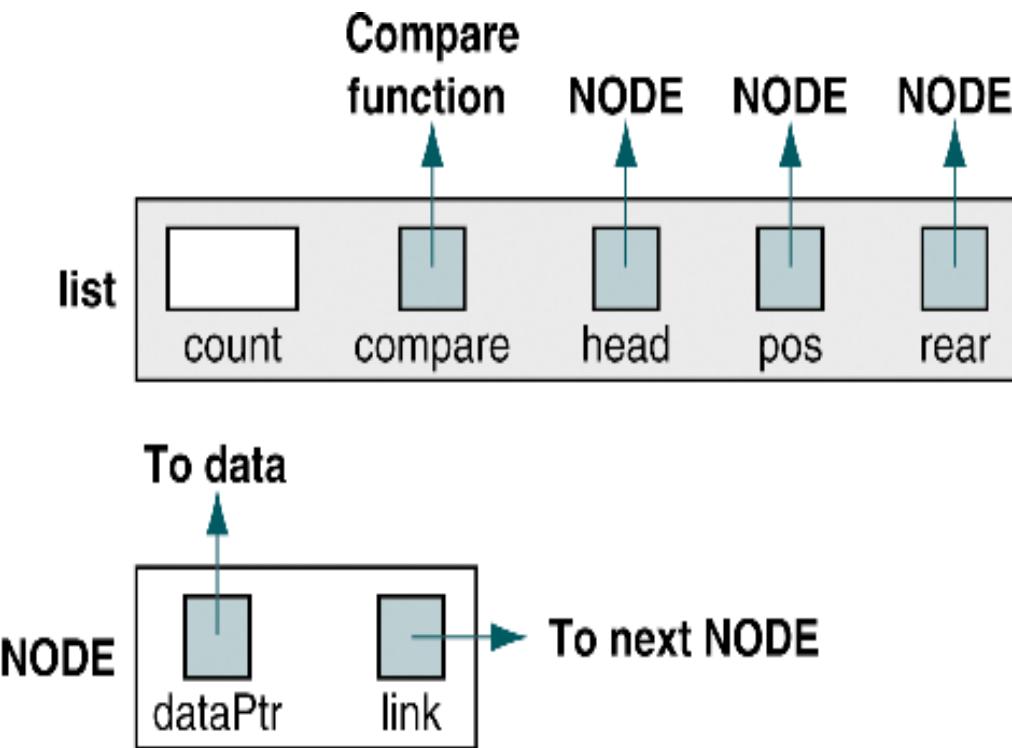
```
Algorithm destroyList (pList)
Deletes all data in list.

    Pre    list is metadata structure to a valid list
    Post   All data deleted
1 loop (not at end of list)
    1 set list head to successor node
    2 release memory to heap
2 end loop
    No data left in list. Reset metadata.
3 set list pos to null
4 set list count to 0
end destroyList
```

5-3 List ADT

We begin with an extensive discussion of the list data structure and the compare function that is required to maintain the list in sequence. We then develop the C functions required to implement the ADT.

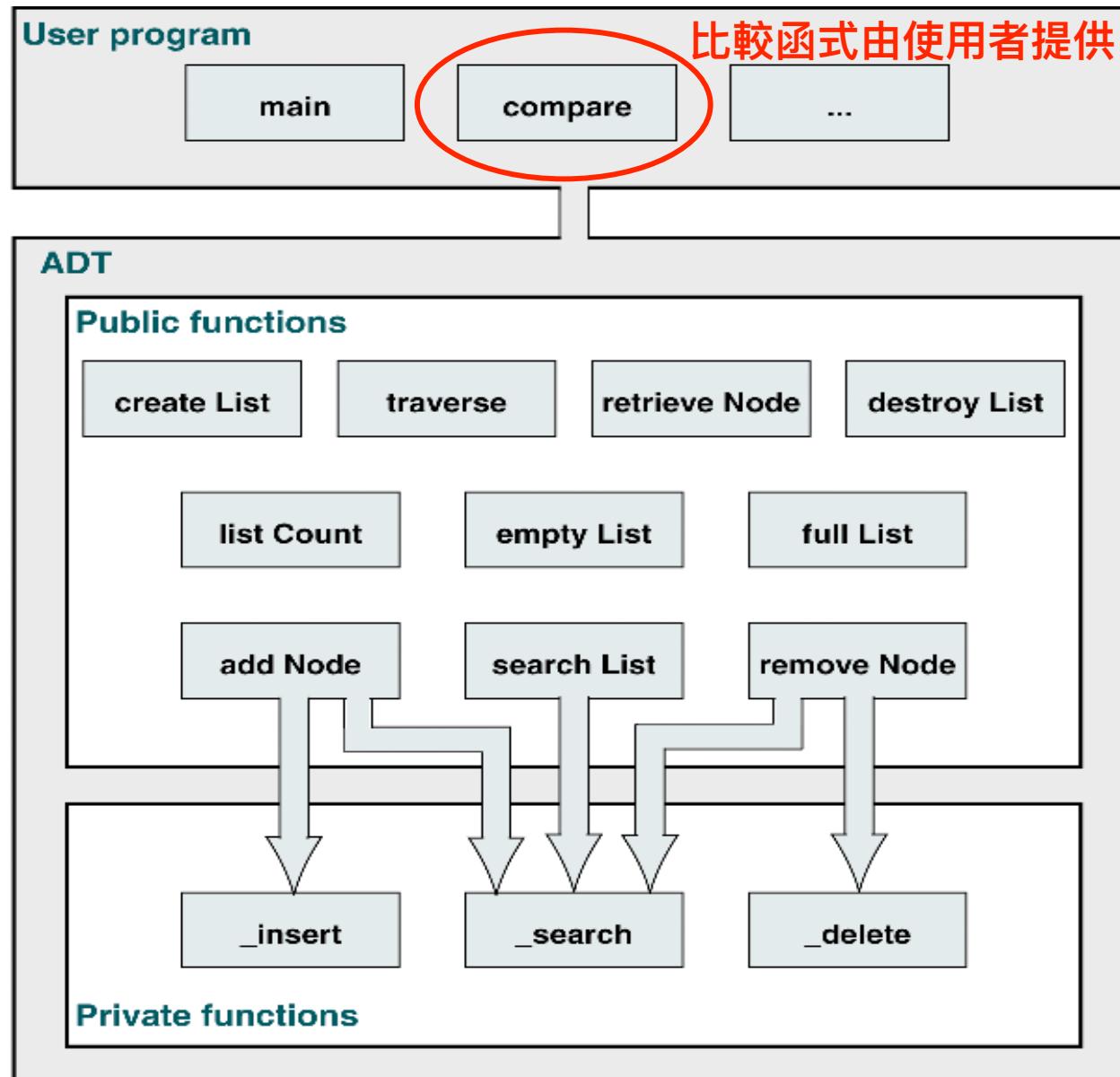
ADT Structure



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* arg1, void* arg2);
15    } LIST;
```

List ADT Functions



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);
```

List ADT Prototype Declarations (cont.)

```
15    bool  retrieveNode (LIST*  pList,
16                      void*  pArgu,
17                      void** dataOutPtr);
18
19    bool  traverse      (LIST*  pList,
20                      int    fromWhere,
21                      void** dataOutPtr);
22
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 in C

```
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
```

Create List in C (cont.)

```
8  */
9  LIST* createList
10     (int (*compare) (void* arg1, 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
```

Internal Insert Function in C

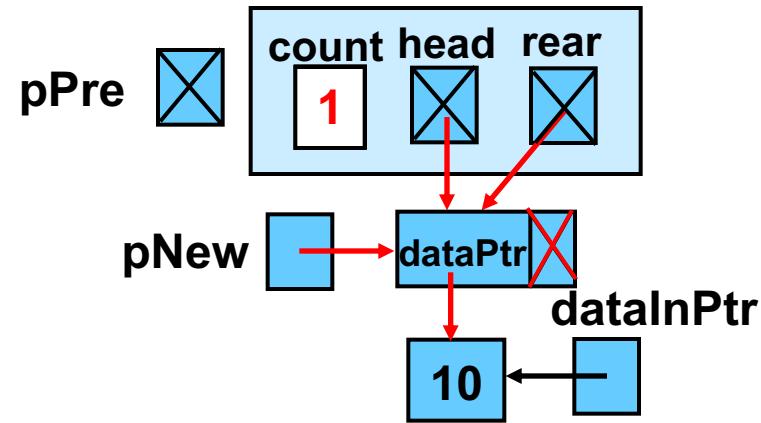
```
1  /*===== _insert ======*
2   Inserts data pointer into a new node.
3   Pre    pList pointer to a valid list
4           pPre   pointer to data's predecessor
5           dataInPtr data pointer to be inserted
6   Post   data have been inserted in sequence
7   Return boolean, true if successful,
8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

Internal Insert Function (continued)

```
9  */
10 static bool _insert (LIST* pList, NODE* pPre,
11                      void* dataInPtr)
12 {
13     //Local Definitions
14     NODE* newNode;
```

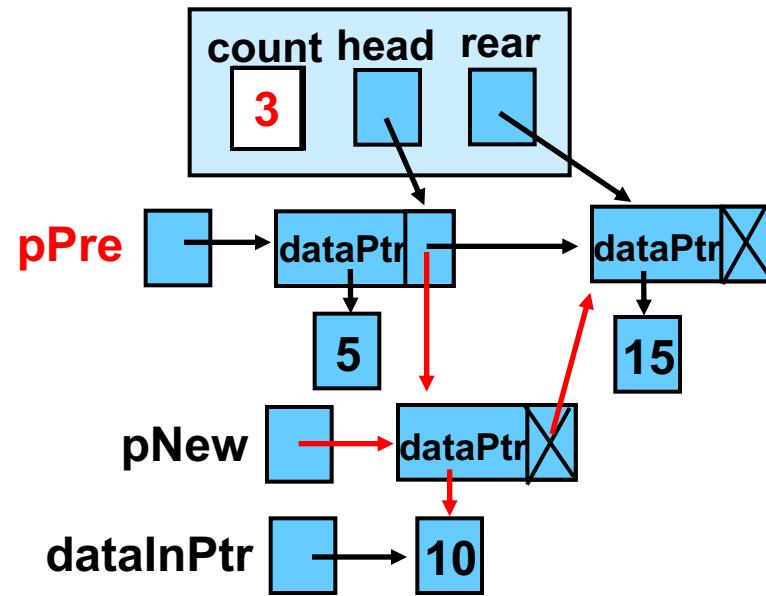
Internal Insert Function in C (cont.)

```
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) insert在第一位  
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
```



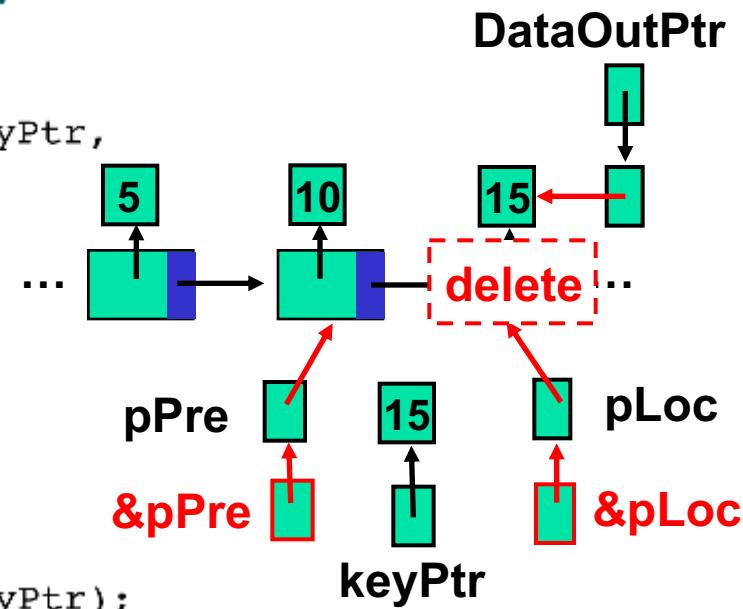
Internal Insert Function (cont.)

```
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 in C

```
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
```

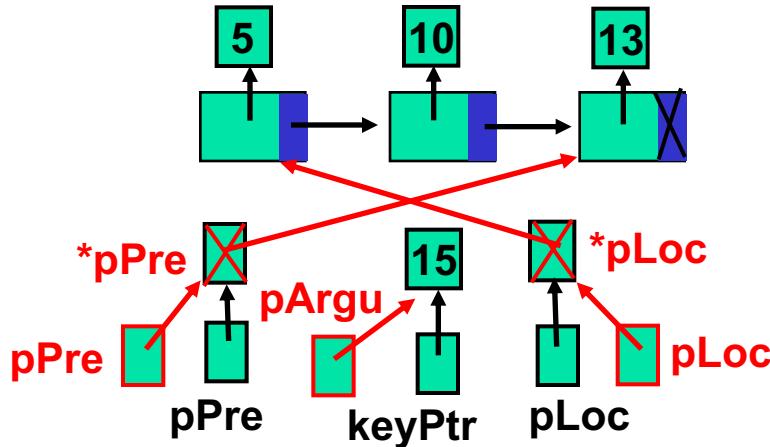


Internal Search Function in C

```
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) double pointer
17 {
18 //Macro Definition
19 #define COMPARE \
20   ( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )
```

Internal Search Function in C (cont.)

```
21 //define COMPARE_LAST \
22     ((* pList->compare) (pArgu, pList->rear->dataPtr))
23
24 //Local Definitions
25     int result;
26
27 //Statements
28     *pPre = NULL;
29     *pLoc = pList->head;
30     if (pList->count == 0)
31         return false;
32
33
34 // Test for argument > last node in list
35 if (COMPARE_LAST > 0) worst case變best case (一次解決)
36 {
37     *pPre = pList->rear;
38     *pLoc = NULL;
39     return false;
40 } // if
```



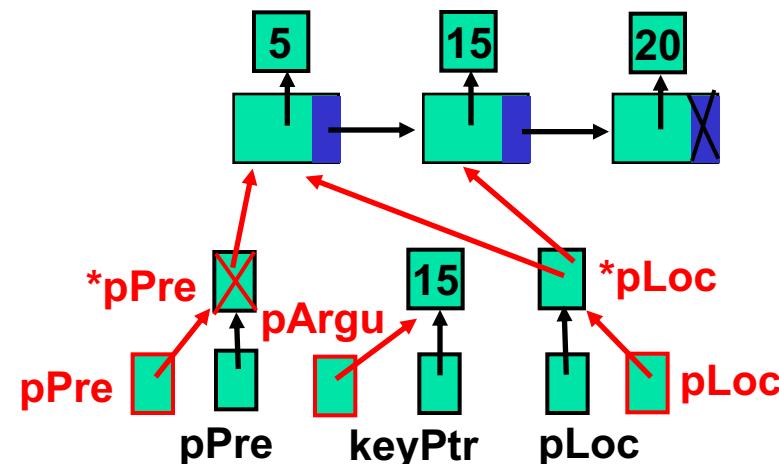
rear pointer的用處

Internal Search Function in C (cont.)

```
#define COMPARE \
( ((* pList->compare) (pArgu, (*pLoc)->dataPtr)) )
```

Note:

```
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
```

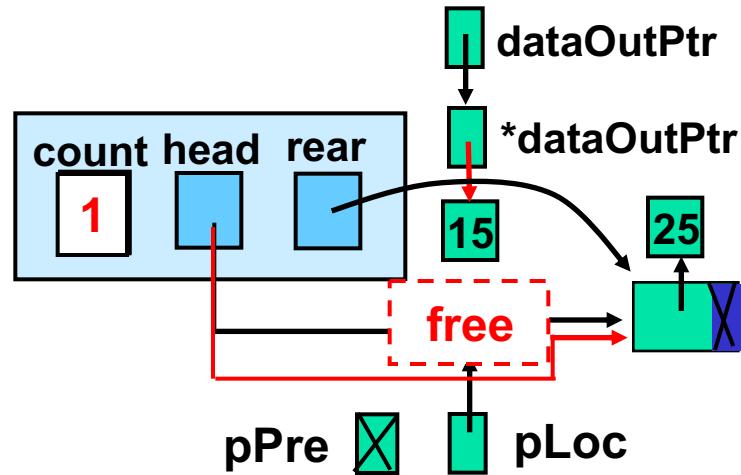
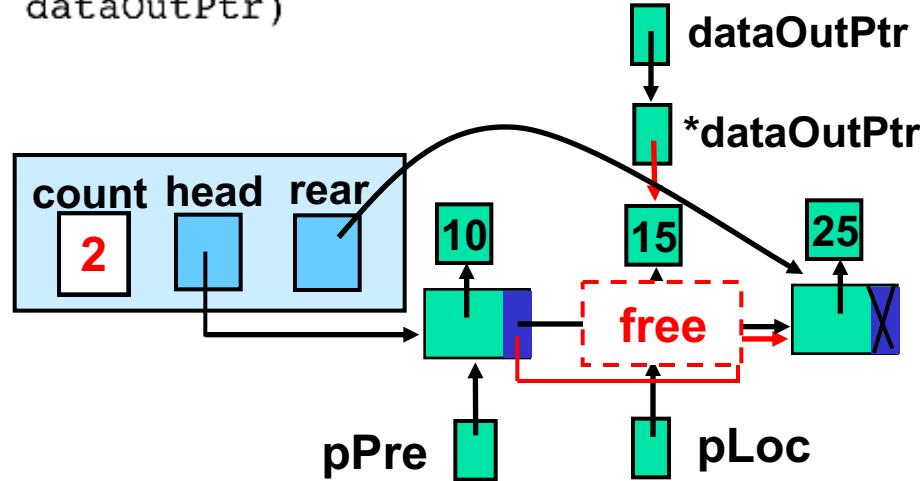


Internal Delete Function in C

```
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
```

Internal Delete Function in C (cont.)

```
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 User Interface in C

```
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 */
```

Search User Interface in C (cont.)

```
 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
```

Retrieve Node in C

```
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
```

Empty List in C

```
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 in C

```
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     free (temp);
17     return false;
18 } // if
19 // Dynamic memory full
20 return true;
21
22
23
24 } // fullList
```

List Count in C

```
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 List in C

Traverse List

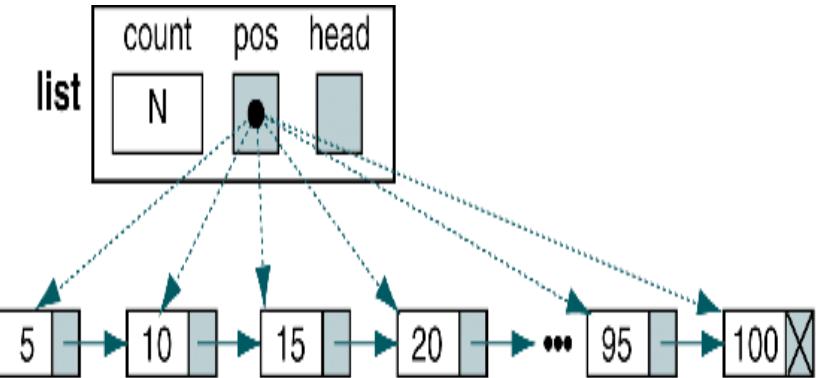
```
1  /*===== traverse ======*/
2  Traverses a list. Each call either starts at the
3  beginning of list or returns the location of the
4  next element in the list.
5  Pre    pList      pointer to a valid list
6          fromWhere  0 to start at first element
7          dataPtrOut address of pointer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

Traverse List (*continued*)

```
9      Return true node located; false if end of list
10 */
11 bool traverse (LIST* pList,
12                 int    fromWhere,
```

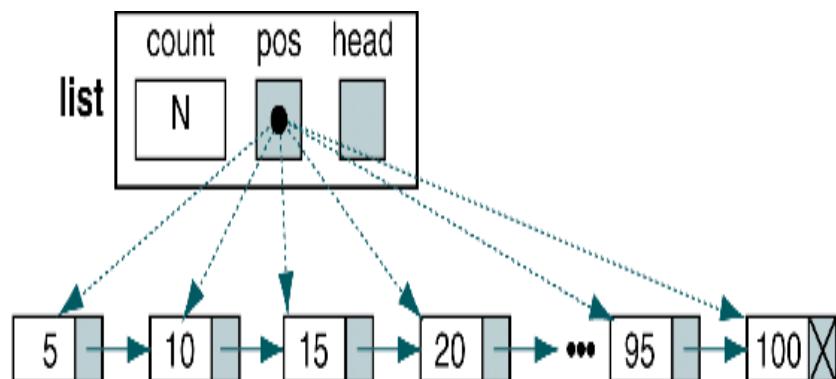
Traverse List in C (cont.)

```
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
```



Traverse List in C (cont.)

```
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 in C

```
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        {
```

Destroy List in C (cont.)

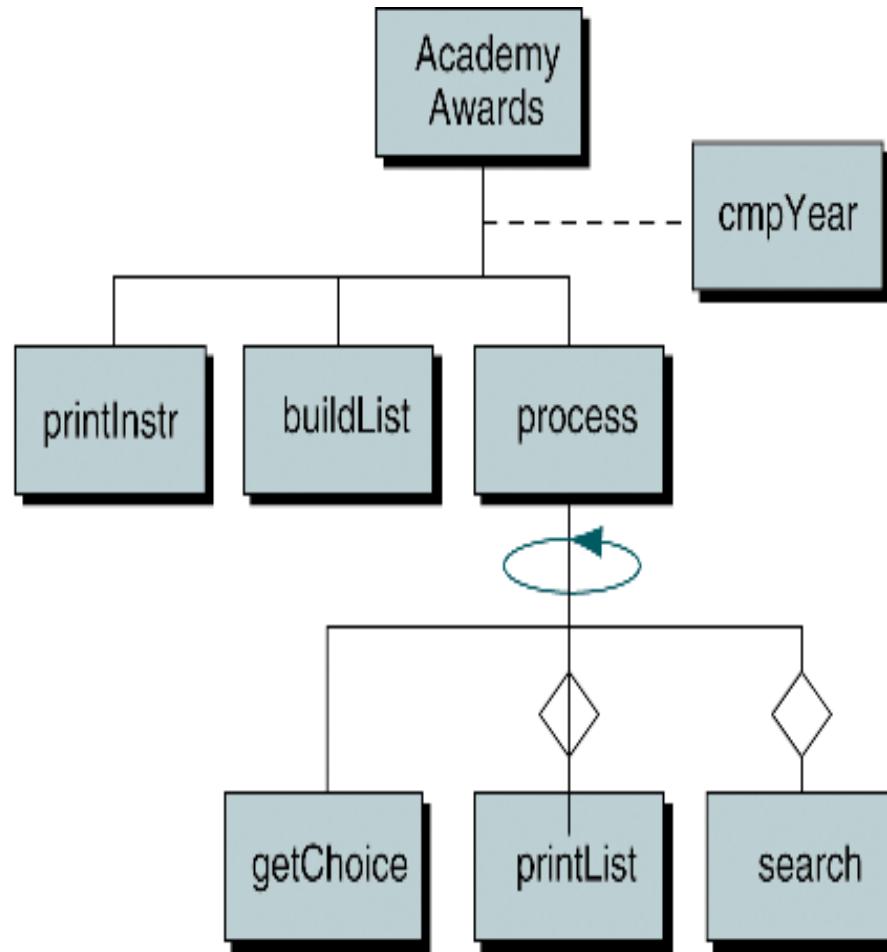
```
17         // First delete data
18         free (pList->head->dataPtr);
19
20         // Now delete node
21         deletePtr      = pList->head;
22         pList->head    = pList->head->link;
23         pList->count--;
24         free (deletePtr);
25     } // while
26     free (pList);
27 } // if
28 return NULL;
29 } // destroyList
```

5-4 Application

To demonstrate how easily we can implement a list once we have an abstract data type, we implement a list of award-winning pictures and their directors. We then develop eight functions to implement the application. We conclude with a discussion of the basic steps required to test a list implementation.

- Data Structure
- Application Functions
- Testing Insert and Delete Logic

Academy Awards List Design



Data Structure for Academy Awards

```
1  /*Data Structure for Academy Awards
2      Written by:
3      Date:
4  */
5
6  const short STR_MAX = 41;
7
8  typedef struct
9  {
10     short    year;
11     char     picture [STR_MAX];
12     char     director[STR_MAX];
13 } PICTURE;
```

Machine for Academy Awards

```
1  /*This program maintains and displays a list of
2   Academy Awards Motion Pictures.
3       Written by:
4       Date:
5   */
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <cType.h>
9 #include <stdbool.h>
10 #include "P5-16.h"           // Data Structure
11 #include "linkListADT.h"
12
13 //Prototype Declarations
14 void printInstr (void);
15 LIST* buildList (void);
16 void process (LIST* list);
17 char getChoice (void);
18 void printList (LIST* list);
19 void search (LIST* list);
20
```

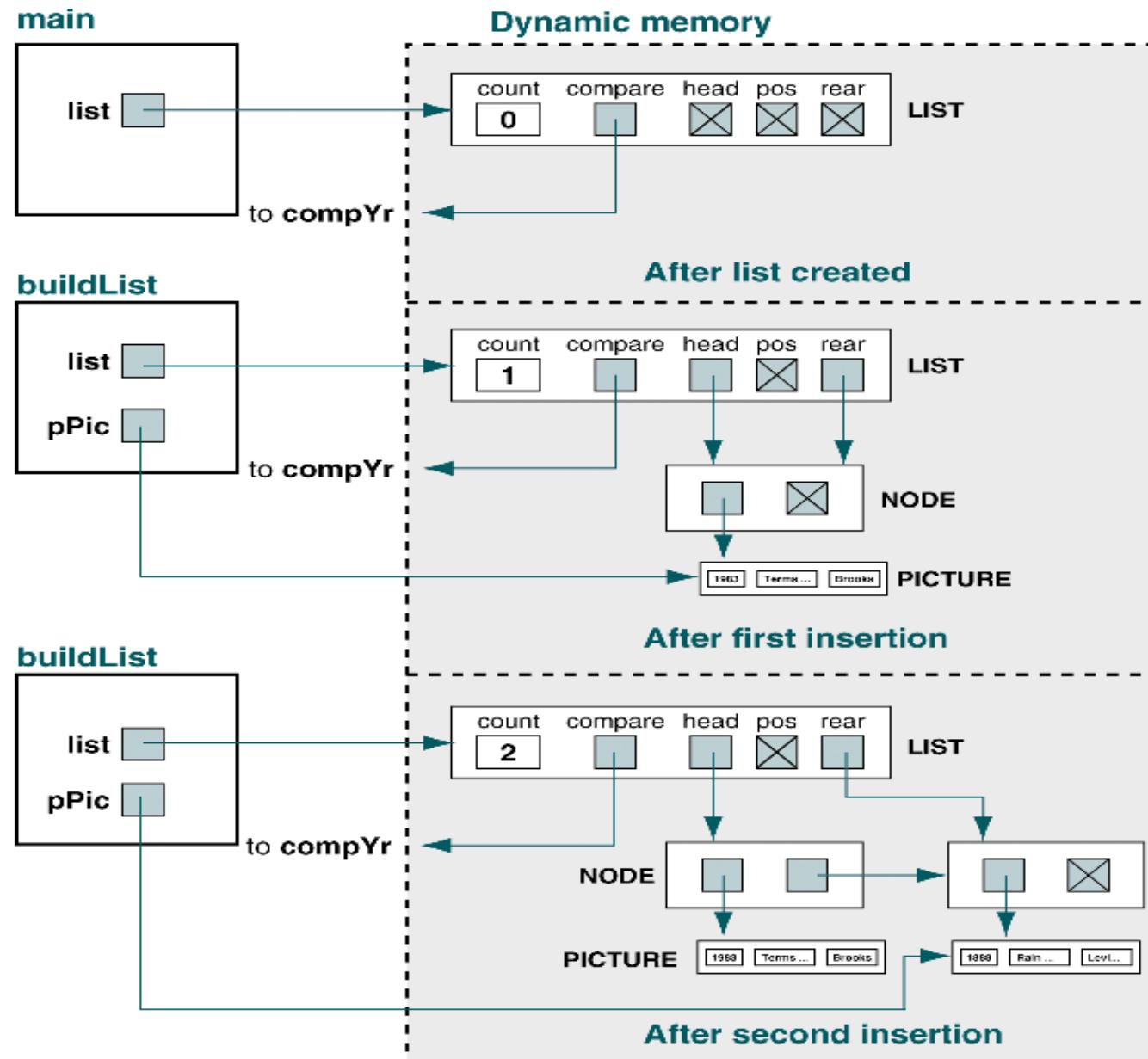
Machine for Academy Awards (cont.)

```
21     int    cmpYear    (void* pYear1, void* pYear2);
22
23 int main (void)
24 {
25 // Local Definitions
26     LIST* list;
27
28 // Statements
29     printInstr ();
30     list = buildList ();
31     process (list);
32
33     printf("End Best Pictures\n"
34             "Hope you found your favorite!\n");
35     return 0;
36 } // main
```

Print Instructions for User

```
1  /*----- printInstr -----*/
2  Print instructions to user.
3      Pre    nothing
4      Post   instructions printed
5  */
6 void printInstr (void)
7 {
8 //Statements
9     printf("This program prints the Academy Awards \n"
10        "Best Picture of the Year and its director.\n"
11        "Your job is to enter the year; we will do\n"
12        "the rest. Enjoy.\n");
13    return;
14 } // printInstr
```

Build List Status after 2nd Insert



Build List

```
1  /*===== buildList =====*/
2      Reads a text data file and loads the list
3      Pre    file exists in format: yy \t 'pic' \t 'dir'
4      Post   list contains data
5              -or- program aborted if problems
6  */
7  LIST* buildList (void)
8  {
9      //Local Definitions
10     FILE* fpData;
11     LIST* list;
12
13     short yearIn;
14     int addResult;
15
16     PICTURE* pPic;
17
18     //Statements
19     list = createList (cmpYear);
20     if (!list)
21         printf("\aCannot create list\n"),
22             exit (100);
23     fpData = fopen("pictures.dat", "r");
24     if (!fpData)
25         printf("\aError opening input file\n"),
26             exit (110);
```

Build List (cont.)

```
27     while (fscanf(fpData, " %hd", &yearIn) == 1)
28     {
29         pPic = (PICTURE*) malloc(sizeof(PICTURE));
30         if (!(pPic))
31             printf("\aOut of Memory in build list\n"),
32                 exit (100);
33         pPic->year = yearIn;
34
35         // Skip tabs and quote
36         while ((fgetc(fpData)) != '\t')
37             ;
38         while ((fgetc(fpData)) != '''')
39             ;
40         fscanf(fpData, " %40[^\"], %*c", pPic->picture);
41         while ((fgetc(fpData)) != '\t')
42             ;
43         while ((fgetc(fpData)) != '''')
44             ;
```

Build List (cont.)

```
45     fscanf(fpData, " %40[^\"], %*c", pPic->director);
46
47     // Insert into list
48     addResult = addNode (list, pPic);
49     if (addResult != 0)
50         if (addResult == -1)
51             printf("Memory overflow adding movie\n"),
52                 exit (120);
53     else
54         printf("Duplicate year %hd not added\n",
55                pPic->year);
56     while (fgetc(fpData) != '\n')
57         ;
58 } // while
59 return list;
60 } // buildList
```

Process user Choices

```
1  /*===== process =====
2   Process user choices
3   Pre    list has been created
4   Post   all of user's choice executed
5 */
6 void process (LIST* list)
7 {
8 //Local Definitions
9     char choice;
10
11 //Statements
12     do
13     {
14         choice = getChoice ();
15
16         switch (choice)
17         {
18             case 'P': printList (list);
19                         break;
20             case 'S': search (list);
21             case 'Q': break;
22         } // switch
23     } while (choice != 'Q');
24     return;
25 } // process
```

Get User's Choice

```
1  /*===== getChoice =====*/
2  Prints the menu of choices.
3  Pre  nothing
4  Post menu printed and choice returned
5  */
6  char getChoice (void)
7  {
```

Get user's Choice (cont.)

```
8 //Local Definitions
9     char choice;
10    bool valid;
11
12 //Statements
13 printf("===== MENU ===== \n"
14         "Here are your choices:\n"
15         "  S: Search for a year\n"
16         "  P: Print all years  \n"
17         "  Q: Quit          \n\n"
18         "Enter your choice: ");
19 do
20 {
21     scanf(" %c", &choice);
22     choice = toupper(choice);
23     switch (choice)
24     {
25         case 'S':
26         case 'P':
27         case 'Q': valid = true;
28             break;
29         default: valid = false;
30             printf("\aInvalid choice\n"
31                     "Please try again: ");
32             break;
33     } // switch
34 } while (!valid);
35 return choice;
36 } // getChoice
```

Print List

```
1  /*===== printList ======*
2   Prints the entire list
3     Pre    list has been created
4     Post   list printed
5  */
6  void printList (LIST* list)
```

Print List (cont.)

```
7  {
8  //Local Definitions
9  PICTURE* pPic;
10
11 //Statements
12
13 // Get first node
14 if (listCount (list) == 0)
15     printf("Sorry, nothing in list\n\n");
16 else
17 {
18     printf("\nBest Pictures List\n");
19     traverse (list, 0, (void**)&pPic);
20     do
21     {
22         printf("%hd %-40s %s\n",
23                 pPic->year,      pPic->picture,
24                 pPic->director);
25     } while (traverse (list, 1, (void**)&pPic));
26 } // else
27 printf("End of Best Pictures List\n\n");
28 } // printList
```

Search List

```
1  /*===== search ======
2   Searches for year and prints year, picture, and
3   director.
4     Pre    list has been created
5           user has selected search option
6     Post   year printed or error message
7 */
8 void search (LIST* list)
9 {
10 //Local Definitions
11 short      year;
12 bool       found;
13
14 PICTURE   pSrchArgu;
15 PICTURE*  pPic;
16
17 //Statements
18 printf("Enter a four digit year: ");
19 scanf ("%hd", &year);
20 pSrchArgu.year = year;
21
22 found = searchList (list, &pSrchArgu,
23                      (void**) &pPic);
24
25 if (found)
26     printf("%hd %-40s %s\n",
27            pPic->year, pPic->picture, pPic->director);
28 else
29     printf("Sorry, but %d is not available.\n", year);
30 return;
31 } // search
```

Compare Year Function

```
1  /*===== cmpYear =====
2   Compares two years in PICTURE structures
3     Pre  year1 is a pointer to the first structure
4           year2 is a pointer to the second structure
5     Post two years compared and result returned
6     Return -1 if year1 less; 0 if equal; +1 greater
7 */
8  int cmpYear (void* pYear1, void* pYear2)
9  {
10 //Local Definitions
11  int  result;
12  short year1;
13  short year2;
14
```

Compare Year Function (cont.)

```
15 //Statements  
16     year1 = ((PICTURE*)pYear1)->year;  
17     year2 = ((PICTURE*)pYear2)->year;  
18  
19     if (year1 < year2)  
20         result = -1;  
21     else if (year1 > year2)  
22         result = +1;  
23     else  
24         result = 0;  
25     return result;  
26 } // cmpYear
```

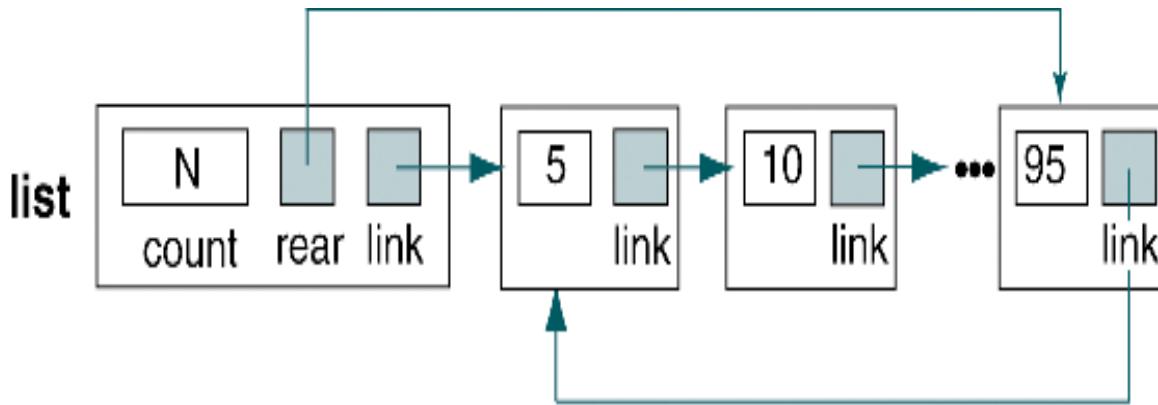
5-4 Complex Implementations

The implementation we have used to this point is known as a single linked list. In this section we introduce three other useful implementation. They are not fully developed. To be included in the list ADT, they would need additional structures, such as a pointer to a compare function and a position pointer as well as functions

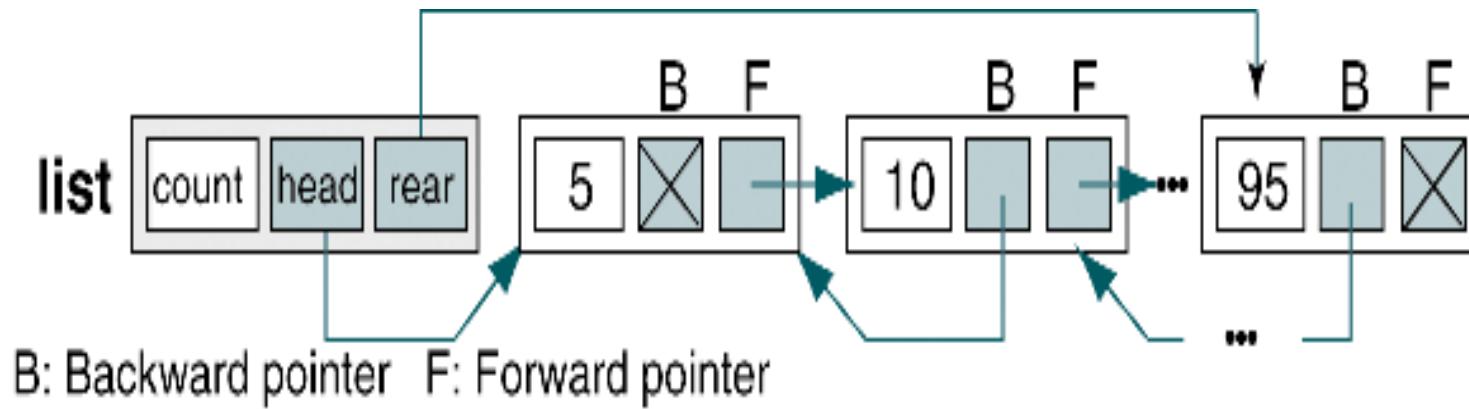
- Circularly Linked list
- Double Linked List
- Multilinked List

Circularly Linked List

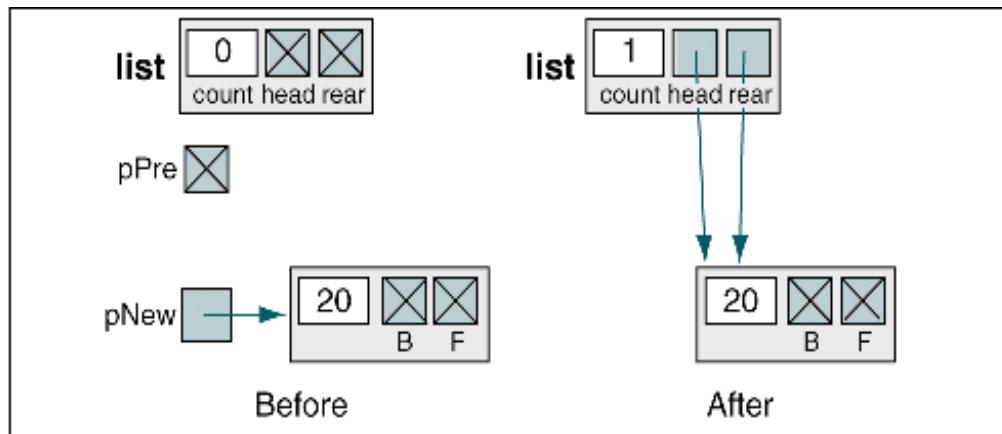
- Circularly linked list are used to allow access to nodes in the middle of the list without starting at the beginning
- If the search target lies before the current node
 - Single linked list: search completes as end of the list is reached
 - Circular list: search continues from the beginning of the list
- If the target does not exist
 - Single linked list: stop search as end of the list is reached
 - Circular list: save the starting node's address and stop search when we have circles around to it



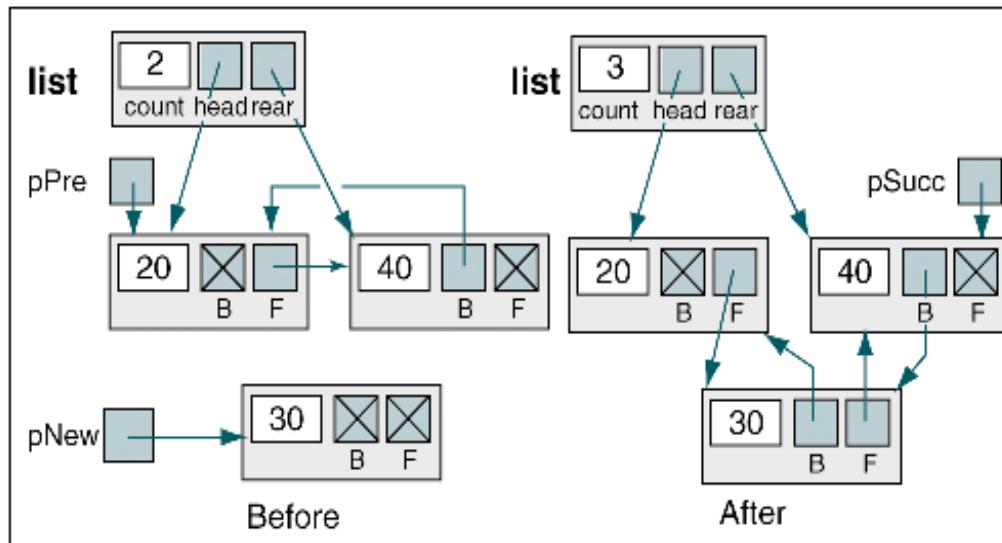
Double Linked List



Double Linked List Insertion



(a) Insert into null list or before first node



(b) Insert between two nodes

Double Linked List Insert Algorithm

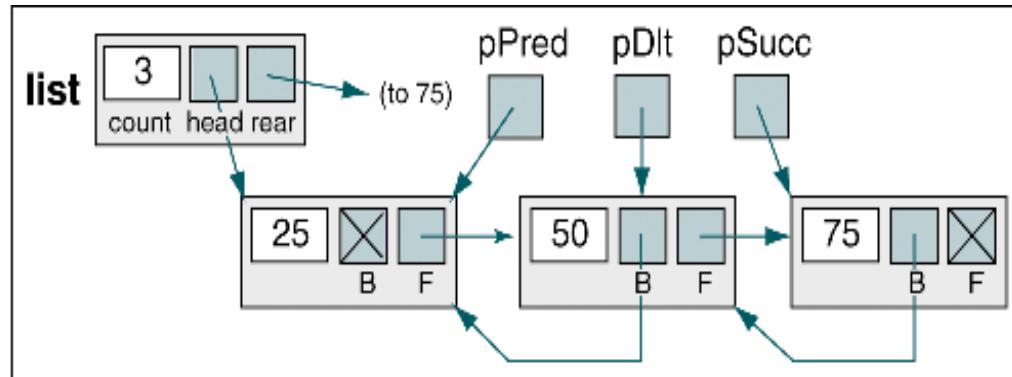
```
Algorithm insertDbl (list, dataIn)
This algorithm inserts data into a doubly linked list.
    Pre    list is metadata structure to a valid list
           dataIn contains the data to be inserted
    Post   The data have been inserted in sequence
    Return 0: failed--dynamic memory overflow
           1: successful
           2: failed--duplicate key presented

1 if (full list)
1 return 0
2 end if
    Locate insertion point in list.
3 set found to searchList
    (list, predecessor, successor, dataIn key)
4 if (not found)
    1 allocate new node
    2 move dataIn to new node
    3 if (predecessor is null)
        Inserting before first node or into empty list
        1 set new node back pointer to null
        2 set new node fore pointer to list head
        3 set list head to new node
    4 else
        Inserting into middle or end of list
        1 set new node fore pointer to predecessor fore pointer
        2 set new node back pointer to predecessor
```

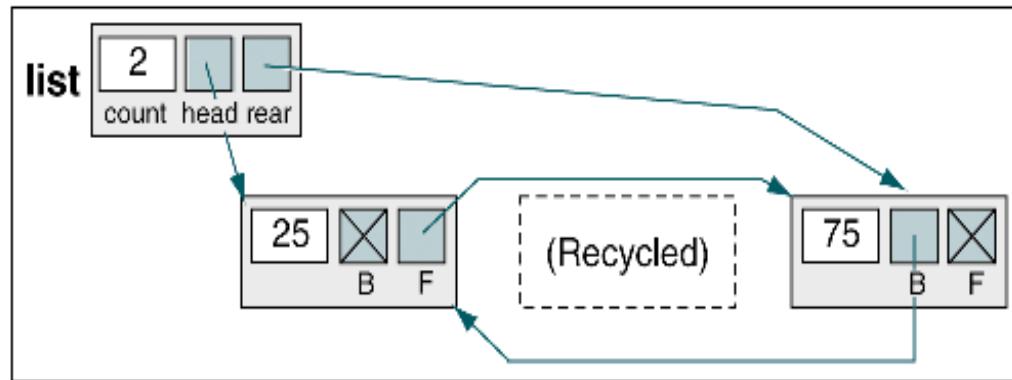
Double Linked List Insert (cont.)

```
5 end if
    Test for insert into null list or at end of list
6 if (predecessor fore null)
    Inserting at end of list--set rear pointer
    1 set list rear to new node
7 else
    Inserting in middle of list--point successor to new
    1 set successor back to new node
8 end if
9 set predecessor fore to new node
10 return 1
5 end if
    Duplicate data. Key already exists.
6 return 2
end insertDbl
```

Double Linked List Delete



(a) Before delete



(b) After deleting 50

Double Linked List Delete Algorithm

```
Algorithm deleteDbl (list, deleteNode)
This algorithm deletes a node from a doubly linked list.
    Pre    list is metadata structure to a valid list
           deleteNode is a pointer to the node to be deleted
    Post   node deleted
1 if (deleteNode null)
    1 abort ("Impossible condition in delete double")
2 end if
3 if (deleteNode back not null)
    Point predecessor to successor
    1 set predecessor      to deleteNode back
    2 set predecessor fore to deleteNode fore
4 else
    Update head pointer
    1 set list head to deleteNode fore
5 end if
6 if (deleteNode fore not null)
    Point successor to predecessor
    1 set successor      to deleteNode fore
    2 set successor back to deleteNode back
7 else
    Point rear to predecessor
    1 set list rear to deleteNode back
8 end if
9 recycle (deleteNode)
end deleteDbl
```

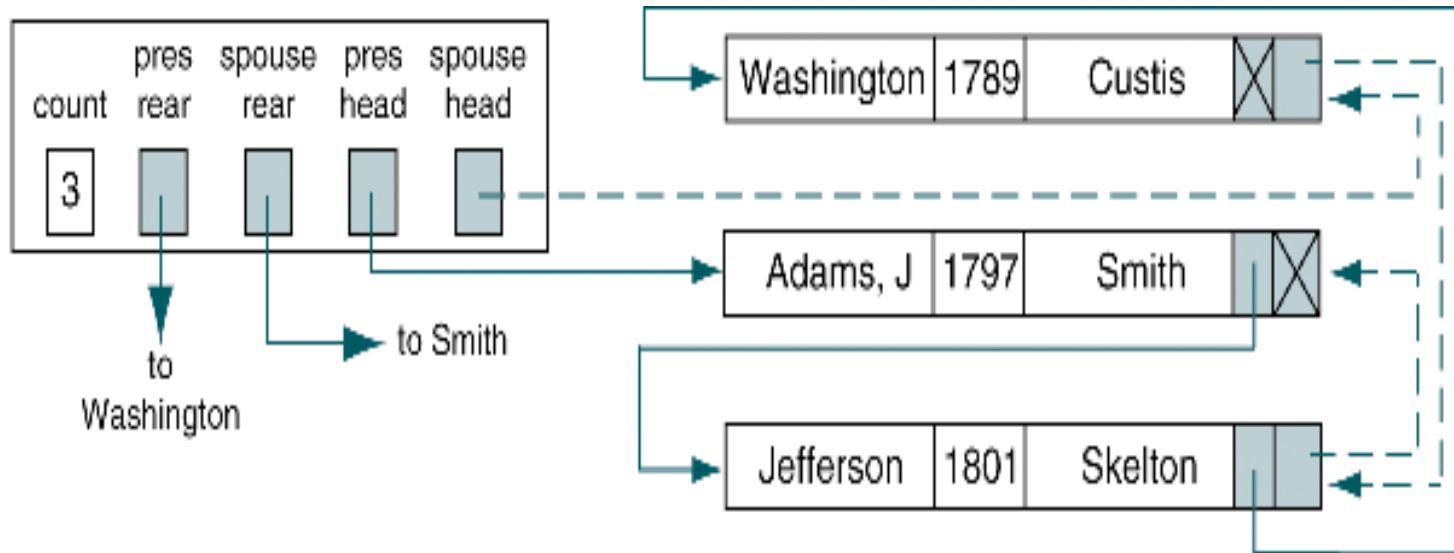
Multilinked List Implementation of Presidents List

President	Year	First lady
Washington, George	1789	Custis, Martha Dandridge
Adams, John	1797	Smith, Abigail
Jefferson, Thomas	1801	Skelton, Martha Wayles
Madison, James	1809	Todd, Dorothy Payne
Monroe, James	1817	Kortright, Elizabeth
Adams, John Quincy	1825	Johnson, Louisa Catherine
Jackson, Andrew	1829	Robards, Rachel Donelson
Van Buren, Martin	1837	Hoes, Hannah
Harrison, William H.	1841	Symmes, Anna
Tyler, John	1841	Christian, Letitia

Multilinked List Implementation of Presidents List (cont.)

- In multilinked list the same set of data can be processed in multiple sequence, while the data are not replicated
- In the following example, two logical lists in the one physical list: one for president and one for the spouse

只增加pointer，沒有重複儲存資料



Multilinked List Add Node

