Dynamic Data Structures with C

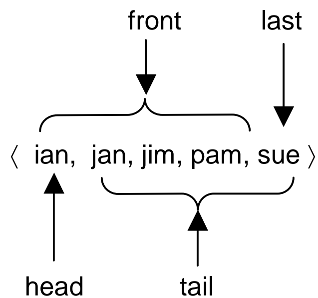
Terry Marris July 2010

5 Lists

We have looked at stacks and queues, where data is added and removed from their ends. We now look at lists. Items are inserted anywhere in a list and removed from anywhere in a list. The items in a list are held in some kind of order e.g. by position, by time, in alphabetical order. A familiar application of lists may be found in word processors where the user determines the order of the characters typed, and inserts and deletes characters from any position in the text.

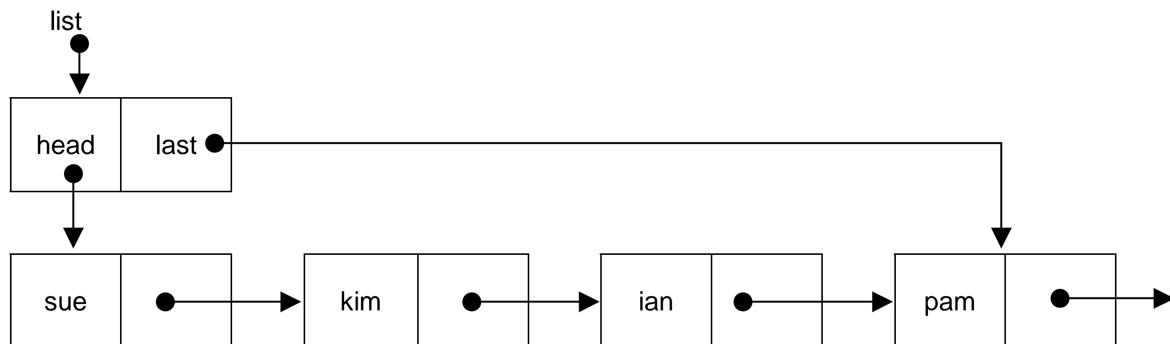
5.1 List

A list is a sequence of data items together with operations to add data to and to remove data from the list. We define the usual terms on a sequence:



The *head* refers to the first item in the sequence, *tail* is the sequence without the head. *Last* refers to the last item in the sequence, *front* to the sequence without the last item.

A diagram representing a list is shown below.

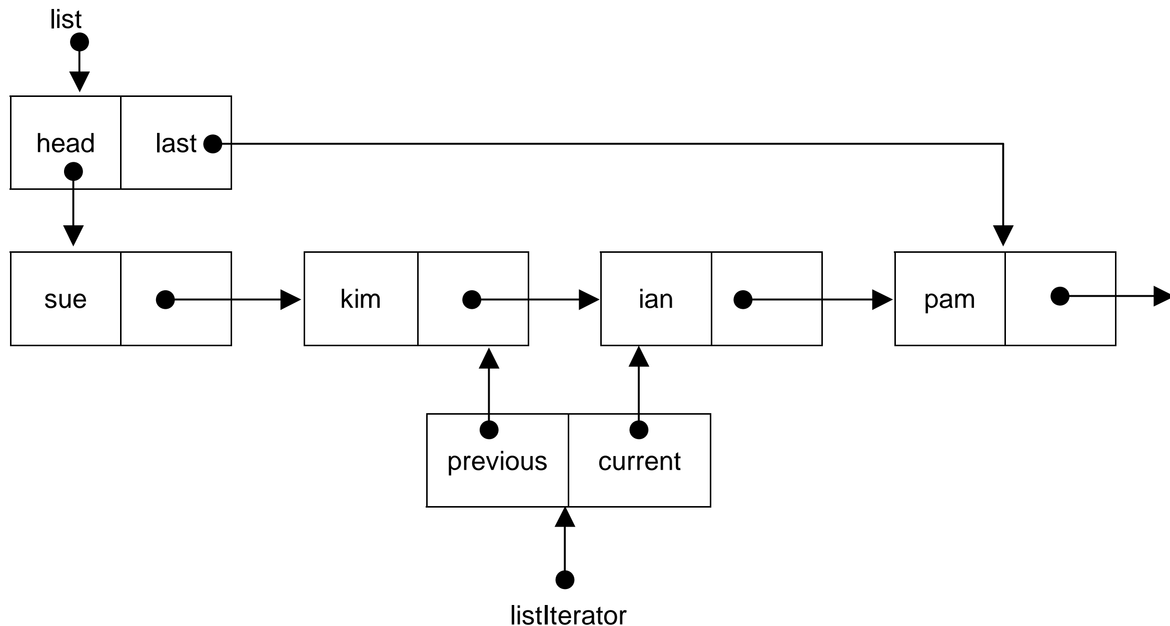


A *list* is a pointer to a list structure. The *head* points to the first node in the list, *last* points to the last node in the list.

5.2 List Iterator

A list iterator is a mechanism that allows you to visit the elements in a list. With a list iterator you can add items, remove items, and visit each item in order from *head* to *last*. You can have several list iterators accessing the elements of a list for reading only at any one time, but only one iterator at a time that updates a list. Imagine the chaos that could happen if you had two iterators simultaneously changing the contents of a list .

A diagram representing a list iterator is shown below.



A list iterator is a pointer to a *ListIterator* structure. It has two members, *previous* and *current.* Both *previous* and *current* are pointers to list nodes. The iterator functions move the pointers through the list. *current* points to the node of interest. *previous* marks the last position of *current*. You can insert new nodes immediately before *current*, remove the node pointed to by *current*, and use *current* to visit each node in turn.

5.3 List Interface

We introduce the interface to our *List* type. We specify *List* as a pointer to a *ListHeader* and a *ListIterator* as a pointer to a *ListIteratorStruct*. *ListHeader* and *ListIteratorStruct* are both defined in the implementation.

```
/* list.h */

#ifndef LIST
#define LIST

typedef struct ListHeader *List;
typedef struct ListIteratorStruct *ListIterator;
typedef void *(*Duplicate)(void *);

/* listError: reports list errors, halts program execution */
int listError(const char *e);
```

```
/* newList: returns a new, empty list */
List newList(Duplicate);

/* disposeList: deallocates memory occupied by List */
List disposeList(List *list);

/* isEmptyList: returns 0 if list is empty */
int isEmptyList(const List list);

/* addListItem: adds data item onto last end of list */
List addListItem(void * const data, List list);

/* newListIterator: returns a new iterator for the given list. */
ListIterator newListIterator(const List list);

/* listIteratorHasNext: returns true if the list has an item not yet
visited in the current iteration */
int listIteratorHasNext(const ListIterator iter);

/* listIteratorNext: returns the next item in the current iteration
*/
void *listIteratorNext(ListIterator iter);

/* listIteratorAdd: inserts item before the current iterator position
*/
List listIteratorAdd(void *data, ListIterator iter);

/* listIteratorRemove: removes and returns the last item retrieved by
listIteratorNext */
List listIteratorRemove(ListIterator iter);

/* disposeListIterator: deallocates memory occupied by ListIterator
*/
ListIterator disposeListIterator(ListIterator *iter);

#endif
```

Given this interface we can perform the usual list operations.

Create a new list:

```
  List list = newList((Duplicate)copyChar);
```

where *copyChar()* is a function, supplied by the user, to make a duplicate copy of a stored element.

Add a data item onto the end of a list:

```
char chA = 'A';
addListItem(&chA, list);
```

Use the iterator methods, *newListIterator()*, *listIteratorHasNext()* and *listIteratorNext()* to visit each node in turn and print its data:

```
ListIterator iter = newListIterator(list);
while (listIteratorHasNext(iter)) {
  char ch = *(char *)listIteratorNext(iter);
  printf("%c ", ch);
}
```

The line

```
ListIterator iter = newListIterator(list);
```

connects an iterator with its list.

We loop for as long as the iterator has a node not yet visited

```
while (listIteratorHasNext(iter))
```

and get the next item visited by the iterator.

```
char ch = *(char *)listIteratorNext(iter);
```

Use the list iterator to add an item to the front of the list:

```
ListIterator iter = newListIterator(list);
listIteratorAdd(&chA, iter);
```

Use the list iterator to add an item to the rear of the list:

```
iter = newListIterator(list);
while (listIteratorHasNext(iter))
  listIteratorNext(iter);
listIteratorAdd(&chA, iter);
```

Use the list iterator to add an item, *X* before a chosen one, *D*, in the list:

```
char chX = 'X';
iter = newListIterator(list);
while (listIteratorHasNext(iter)) {
  char ch = *(char *)listIteratorNext(iter);
  if (ch == 'D')
    listIteratorAdd(&chX, iter);
  }
```

Remove the first item in a list:

```
iter = newListIterator(list);
listIteratorNext(iter);
listIteratorRemove(iter);
```

listIteratorRemove() removes the item last visited by a call to *listIteratorNext()*.

Remove the last item in a list:

```
iter = newListIterator(list);
while (listIteratorHasNext(iter))
  listIteratorNext(iter);
listIteratorRemove(iter);
```

Remove a chosen item from a list:

```
iter = newListIterator(list);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  if (ch == 'X')
    listIteratorRemove(iter);
}
```

Release the memory occupied by a list:

```
disposeList(&list);
```

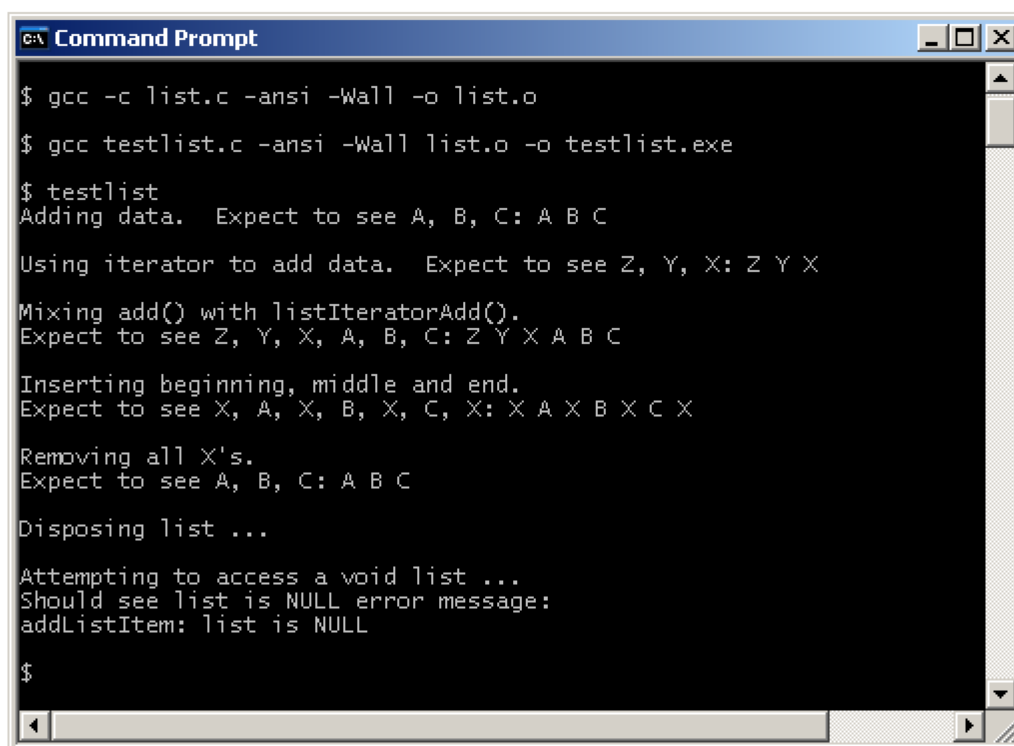
5.4 Testing

We test each function looking for errors. A test program and its run are shown below.

```
/* TestList.c */

#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/* copyChar: returns a duplicate copy of its parameter */
char *copyChar(char *c)
{
  char *t = malloc(sizeof(char));
  t = c;
  return t;
}

int main()
{
  List list = newList((Duplicate)copyChar);
  char chA = 'A';
  char chB = 'B';
  char chC = 'C';
  printf("Adding data.  Expect to see A, B, C: ");
  addListItem(&chA, list);
  addListItem(&chB, list);
  addListItem(&chC, list);
  ListIterator it = newListIterator(list);
  while (listIteratorHasNext(it)) {
    char ch = *(char *)listIteratorNext(it);
    printf("%c ", ch);
  }
  printf("\n\n");

  List list2 = newList((Duplicate)copyChar);
  char chX = 'X';
  char chY = 'Y';
  char chZ = 'Z';
  ListIterator it2 = newListIterator(list2);
  printf("Using iterator to add data.  Expect to see Z, Y, X: ");
  listIteratorAdd(&chX, it2);
  listIteratorAdd(&chY, it2);
  listIteratorAdd(&chZ, it2);
```

```
disposeListIterator(&it2);
it = newListIterator(list2);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  printf("%c ", ch);
}
printf("\n\n");

List list3 = newList((Duplicate)copyChar);
it = newListIterator(list3);
printf("Mixing add() with listIteratorAdd().\n");
printf("Expect to see Z, Y, X, A, B, C: ");
list3 = addListItem(&chA, list3);
listIteratorAdd(&chX, it);
list3 = addListItem(&chB, list3);
listIteratorAdd(&chY, it);
list3 = addListItem(&chC, list3);
listIteratorAdd(&chZ, it);

it = newListIterator(list3);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  printf("%c ", ch);
}
printf("\n\n");

printf("Inserting beginning, middle and end.\n");
it = newListIterator(list);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  if (ch == 'A')
    listIteratorAdd(&chX, it);
  else if (ch == 'B')
    listIteratorAdd(&chX, it);
  else if (ch == 'C')
    listIteratorAdd(&chX, it);
}
list = addListItem(&chX, list);

printf("Expect to see X, A, X, B, X, C, X: ");
it = newListIterator(list);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  printf("%c ", ch);
}
printf("\n\n");

printf("Removing all X's.\n");
it = newListIterator(list);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  if (ch == 'X')
    listIteratorRemove(it);
}

printf("Expect to see A, B, C: ");
it = newListIterator(list);
while (listIteratorHasNext(it)) {
  char ch = *(char *)listIteratorNext(it);
  printf("%c ", ch);
}
```

```
  printf("\n\n");

  printf("Disposing list ...");
  disposeList(&list);
  printf("\n\n");

  printf("Attempting to access a void list ...\n");
  printf("Should see list is NULL error message: ");
  list = addListItem(&chA, list);

  return 0;
}
```



5.5 List Implementation

A list is a sequence of data items. Items are added at any point in the list, and removed from any point.

5.5.1 List Node

A list node is a structure with two members, *data* and *next*.

```
struct ListNodeStruct {
  void *data;
  ListNode next;
};
```

data is defined to be a pointer to *void* so that it can store items of any data type. *next* is a pointer to a node.

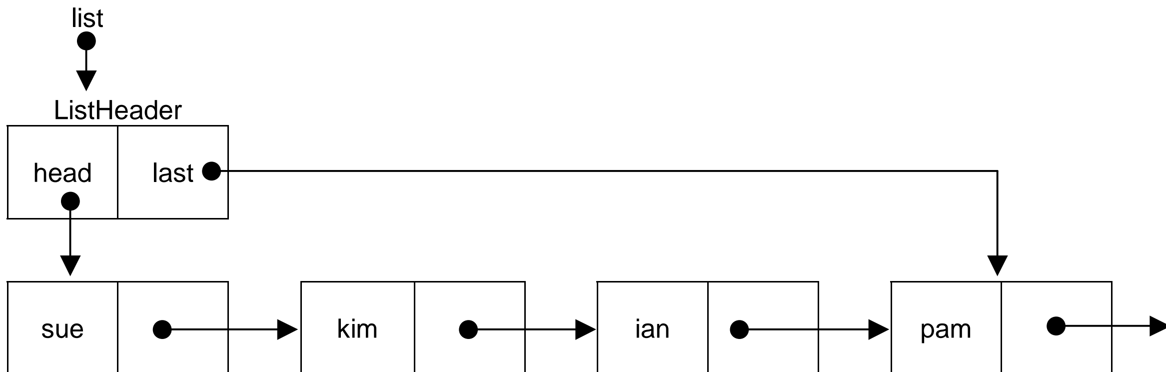
We define *ListNode* as a pointer to a *ListNodeStruct*.

```
typedef struct ListNodeStruct *ListNode;
```

5.5.2 List Header

A list header contains two node pointers, one to the node at the head of the list, one to the last node in the list.

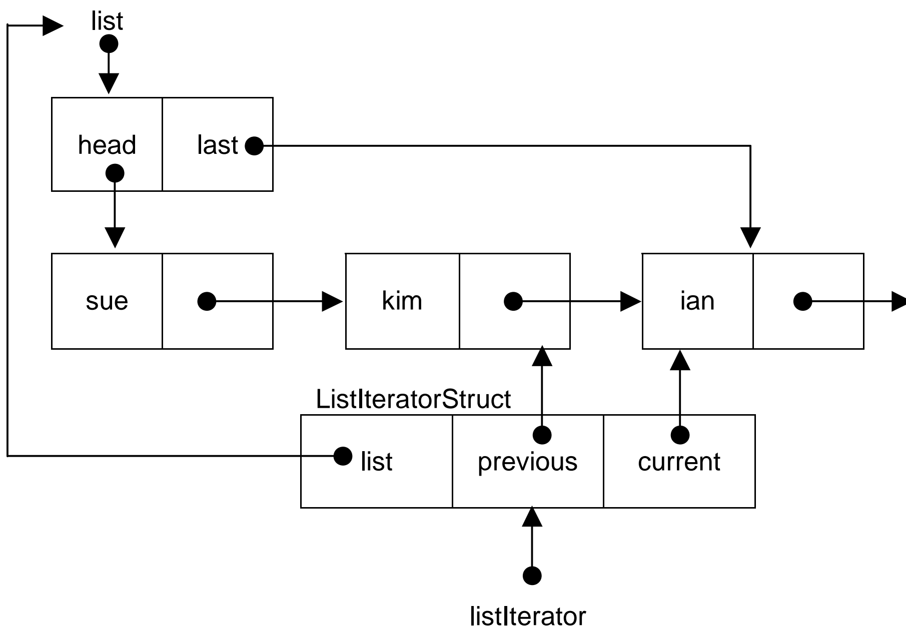
```
struct ListHeader {
  ListNode head;
  ListNode last;
  Duplicate duplicate;
};
```



5.5.3 List Iterator

A list iterator structure contains two node pointers named *previous* and *current*. Their purpose is to mark the current node of interest and to make the addition and deletion of nodes straightforward.

```
struct ListIteratorStruct {
  ListNode previous;
  ListNode current;
  List list;
};
```



Here, the current node of interest is the one with data item *ian*.

5.5.4 List Error

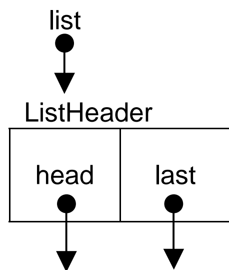
listError() prints the error string parameter then terminates program execution. The function does not return to its caller.

```
/* listError: reports list errors, halts program execution */
int listError(const char *error)
{
  printf("%s\n", error);
  exit(1);
}
```

5.5.5 New List

newList() creates a new, empty list. *head* and *last* are both set to *NULL*.

```
/* newList: returns a new, empty list */
List newList(Duplicate dup)
{
  List list = malloc(sizeof(struct ListHeader));
  if (list == NULL)
    listError("newList: out of memory");
  list->head = NULL;
  list->last = NULL;
  list->duplicate = dup;
  return list;
}
```



5.5.6 Empty List

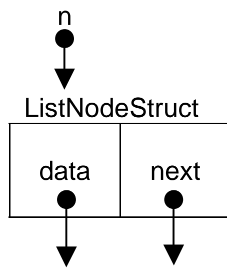
A list is empty if both its *head* and *last* pointers are *NULL*.

```
/* isEmptyList: returns 0 if list is empty */
int isEmptyList(List list)
{
  if (list == NULL)
    listError("isEmptyList: list is NULL");
  return (list->head == NULL && list->last == NULL);
}
```

5.5.7 New List Node

We malloc a hole in memory for a new node, and set both its members, *next* and *data*, to *NULL*.

```
/* newListNode: returns a new, empty list node */
ListNode newListNode()
{
  ListNode n = malloc(sizeof(struct ListNodeStruct));
  if (n == NULL)
    listError("listNode: out of memory");
  n->next = NULL;
  n->data = NULL;
  return n;
}
```



5.5.8 Add List Item

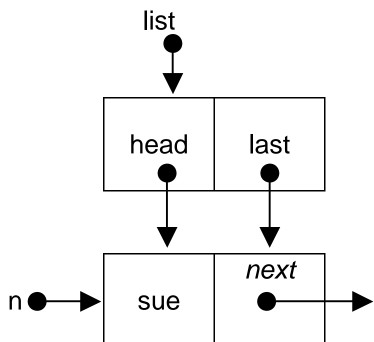
addListItem() adds an item onto the end of the list.

We create a new node and remember that both its members are initially *NULL*. We assign the given data item to *data*.

```
ListNode n = newListNode();
n->data = list->duplicate(data);
```

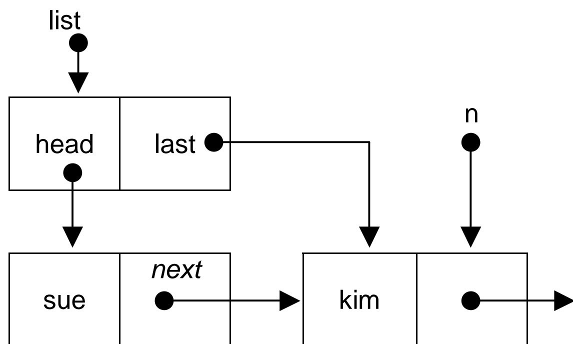
If this is the first node in an empty list we assign the new node to both *head* and *last*.

```
if (isEmptyList(list)) {
  list->head = n;
  list->last = n;
}
```



If this is not the first node we update the *last* node *next* pointer with the new node, and the new node becomes the node pointed to by *last*.

```
list->last->next = n;
list->last = n;
```



The entire function is shown below.

```
/* addListItem: adds an item onto the last end of a list */
List addListItem(void *data, List list)
{
  if (list == NULL)
    listError("addListItem: list is NULL");
  ListNode n = newListNode();
  n->data = list->duplicate(data);
  if (isEmptyList(list)) {
    list->head = n;
    list->last = n;
  }
  else {
    list->last->next = n;
    list->last = n;
  }
  return list;
}
```

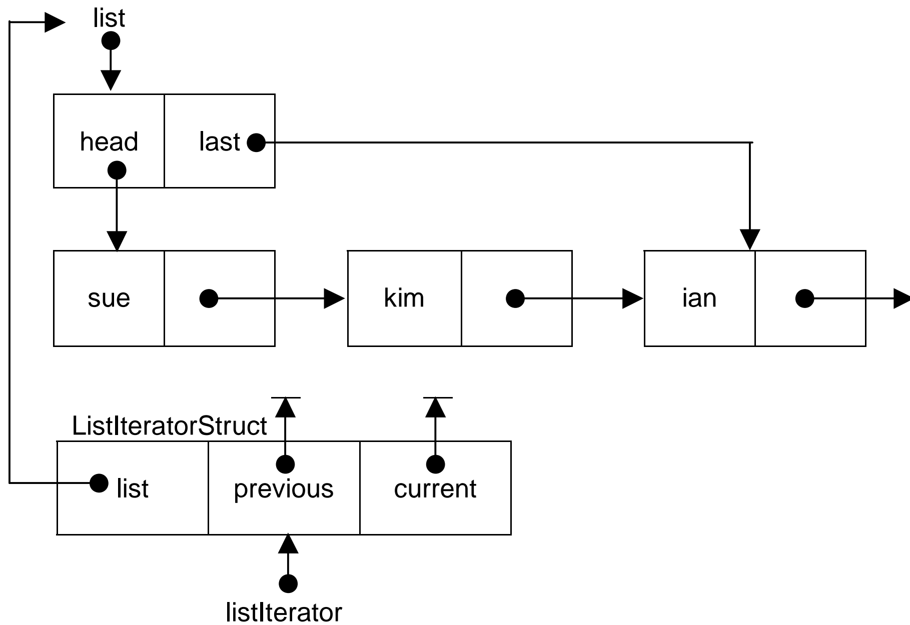
5.5.9 New List Iterator

We malloc a hole in memory for the iterator and set the *previous* and *current* members to *NULL*.

```
/* newListIterator: returns a new list iterator. */
ListIterator newListIterator(List list)
{
  ListIterator iter = malloc(sizeof(struct ListIteratorStruct));
  if (iter == NULL)
    listError("newListIterator: out of memory");
  iter->previous = NULL;
  iter->current = NULL;
  iter->list = list;
  return iter;
}
```

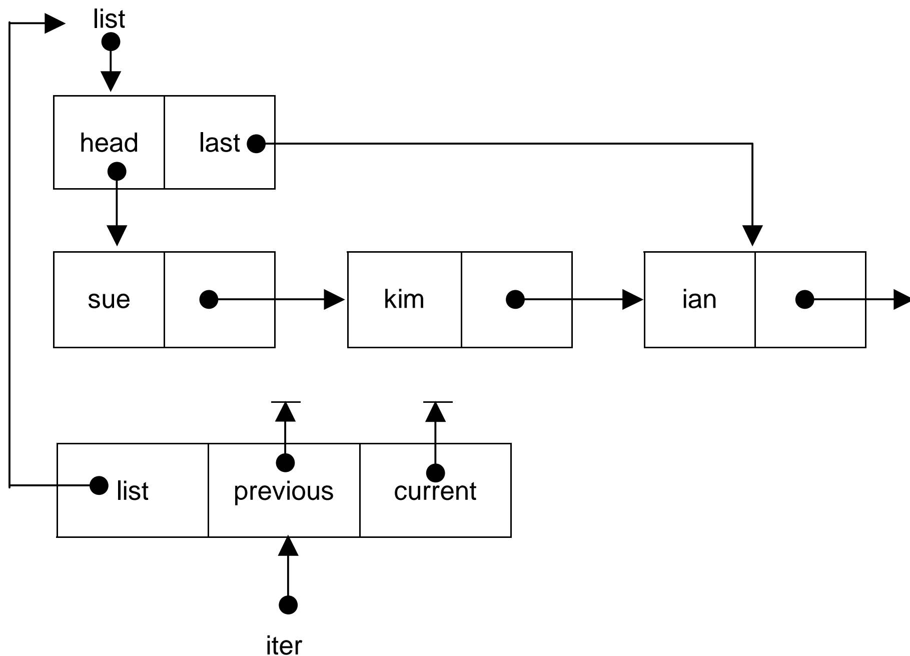
newListIterator() also associates the given list with the iterator.

```
ListIterator newListIterator(List list)
...
iter->list = list;
```



5.5.10 List Iterator Has Next

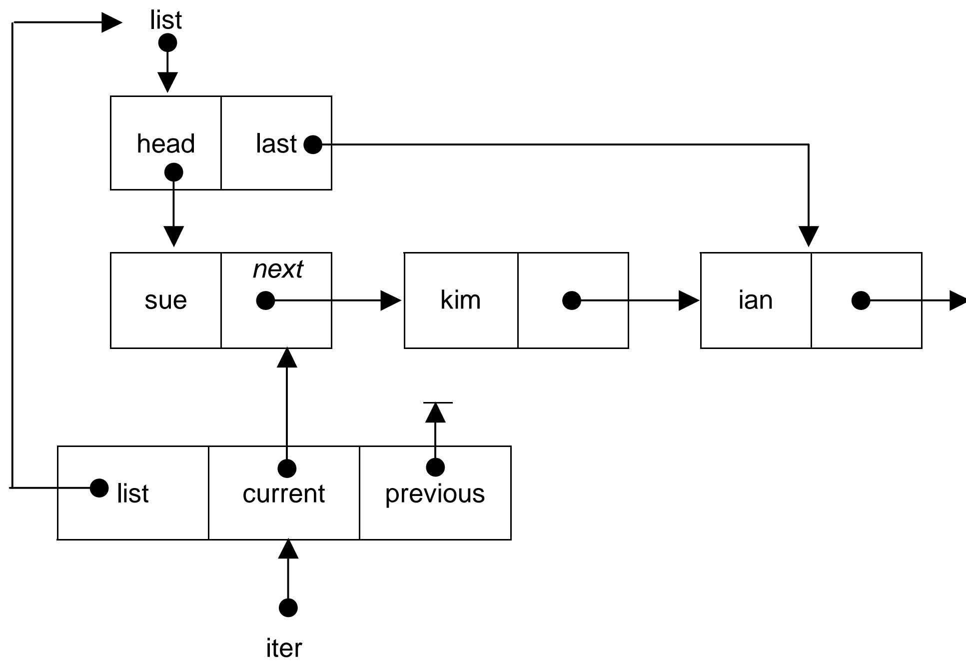
We visit each node in turn, starting with the one at the head of the list. *listIteratorHasNext()* returns true if there is another item to be visited in the current iteration. We start with a newly-initialised iterator.



The head of the list is not *NULL* and so we have at least one node to visit.

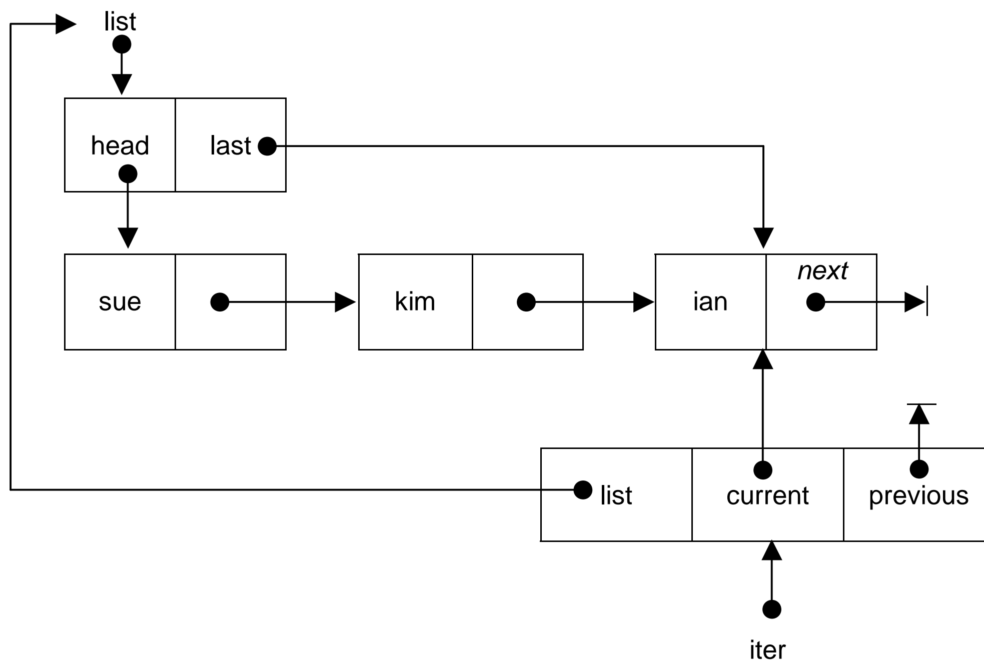
```
if (iter->current == NULL)
  return iter->list->head != NULL;
```

We move the iterator on to the first node.



We can see that *iter->current->next* is not *NULL* and so we have another node to visit.

We move the iterator onto the last node.



Now we can see that *iter->current->next* is *NULL* and so we have no more nodes to visit in this iteration.

The entire function is shown below.

```
/* listIteratorHasNext: returns true if the list has an item not yet
   visited in the current iteration */
int listIteratorHasNext(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorHasNext: list is NULL");
  if (isEmptyList(iter->list))
    return 0;
  if (iter->current == NULL)
    return iter->list->head != NULL;
  return iter->current->next != NULL;
}
```

5.5.11 List Iterator Next

We move the list iterator onto the next node. If we have a newly initialised iterator we move it onto the first node.

```
if (iter->current == NULL)
  iter->current = iter->list->head;
```

If the iterator is already on a node, we move it onto the next one.

```
iter->current = iter->current->next;
```

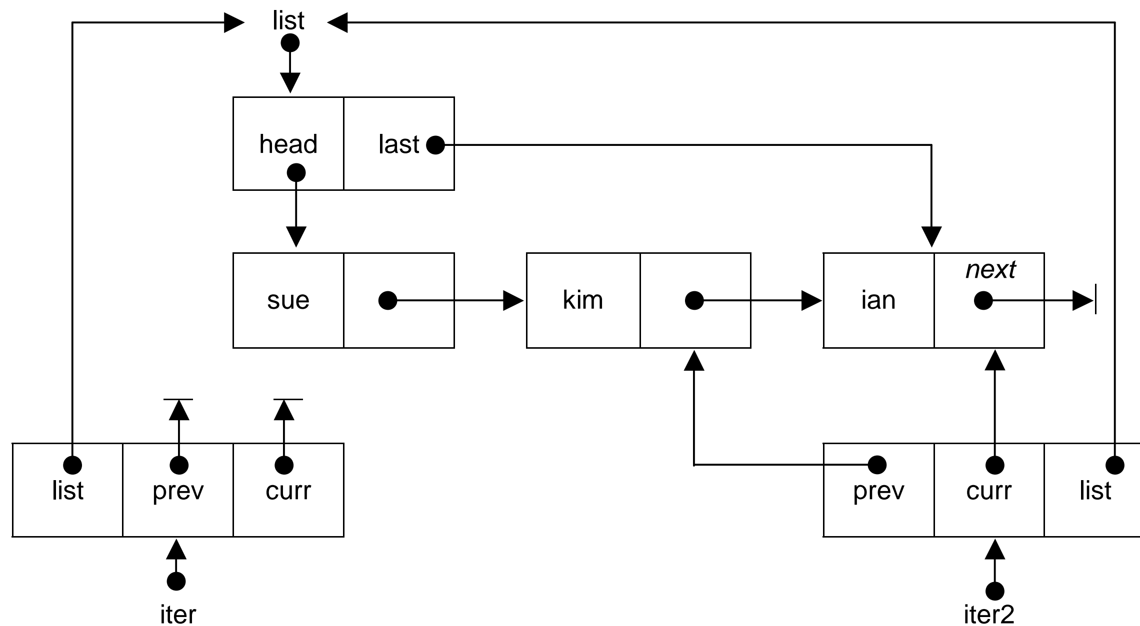
Of course, you should ensure there is a node to be moved on to, perhaps by a call to *listIteratorHasNext()*.

Here is the complete function.

```
/* listIteratorNext: returns the next item in the current iteration
*/
void *listIteratorNext(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorNext: list is NULL\n");
  if (isEmptyList(iter->list))
    listError("listIteratorNext: list is empty");
  if (!listIteratorHasNext(iter))
    listError("listIteratorNext: no next node");
  iter->previous = iter->current;
  if (iter->current == NULL)
    iter->current = iter->list->head;
  else
    iter->current = iter->current->next;
  return iter->list->duplicate(iter->current->data);
}
```

5.5.12 List Iterator Add

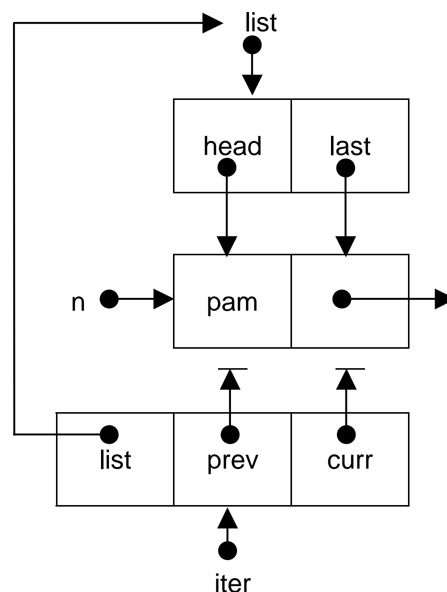
A list iterator ranges in position from before the *head* node to the *last* node.



We have three cases: either we insert into an empty list, or we insert at the front of a non-empty list, or we insert in the middle of a list.

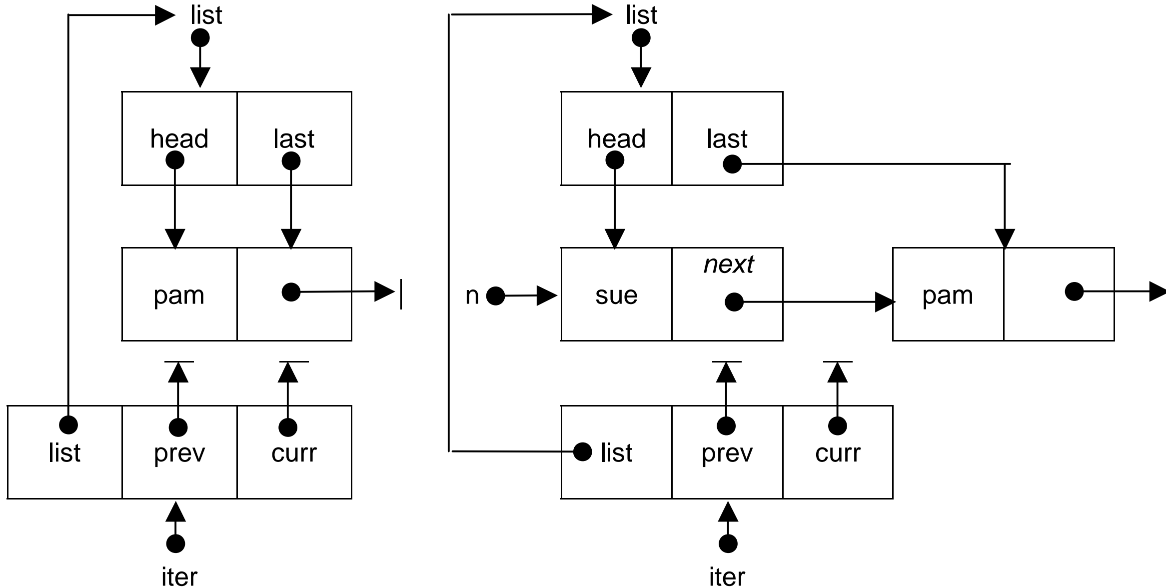
```
if (iter->list->head == NULL)
...
else if (iter->current == NULL)
...
else ...
```

If the list is empty, we update *head* and *last* with pointers to the new node, *n*.



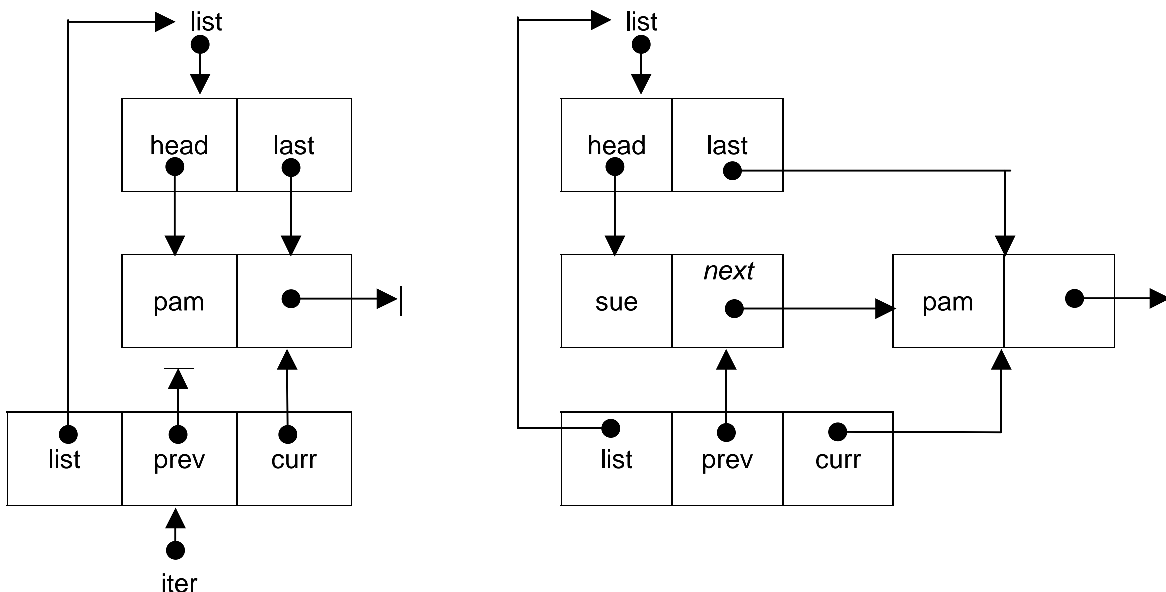
```
if (iter->list->head == NULL) {
    iter->list->head = n;
    iter->list->last = n;
}
```

If the current iterator position is *NULL* we insert a new node, *n*, at the *head* of the list.

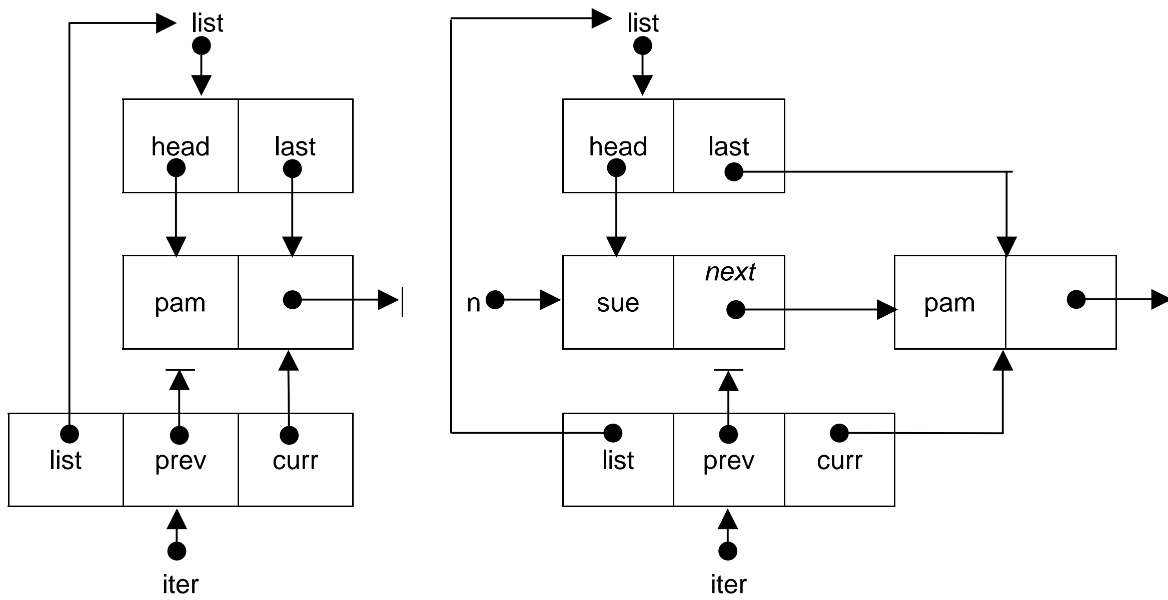



```
if (iter->current == NULL) {
  n->next = iter->list->head;
  iter->list->head = n;
}
```

If we are inserting in the middle of a list, we have two cases to consider: either *current* points to the *head* node and *previous* is *NULL*, or *current* does not point to the *head* node and *previous* is not *NULL*.

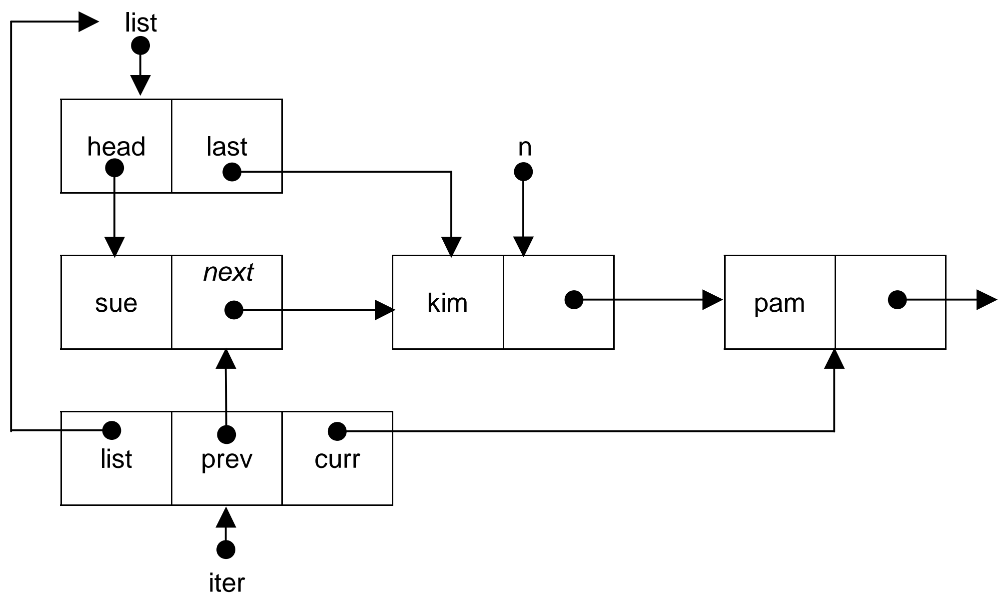


If *head* and *current* both point to the same node, we set the new node, *n*, to point to the *current* one, and update *head* to point to the new node.




```
n->next = iter->current
iter->list->head = n;
```

If *previous* points to a node and *current* to the next node, we update the new node, *n*, to point to the *current* node, and update the *previous* node to point to the new node.




```
n->next = iter->current;
iter->previous->next = n;
```

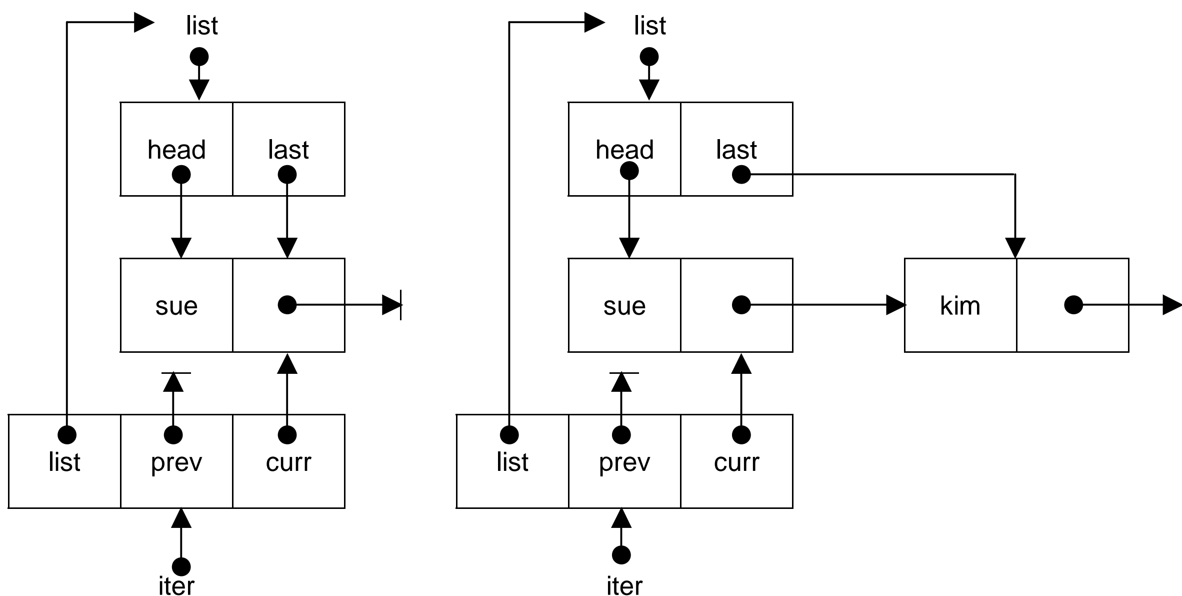
Here is the entire function.

```
/* listIteratorAdd: inserts item before the current iterator position
*/
List listIteratorAdd(void *data, ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorAddItem: list is NULL");
  ListNode n = newListNode();
  n->data = iter->list->duplicate(data);

 if (iter->list->head == NULL) {
    iter->list->head = n;
    iter->list->last = n;
  }
  else if (iter->current == NULL) {
    n->next = iter->list->head;
    iter->list->head = n;
  }
  else {
    n->next = iter->current;
    if (iter->list->head == iter->current)
      iter->list->head = n;
    else if (iter->previous != NULL)
      iter->previous->next = n;
  }
  iter->previous = NULL;
  return iter->list;
}
```

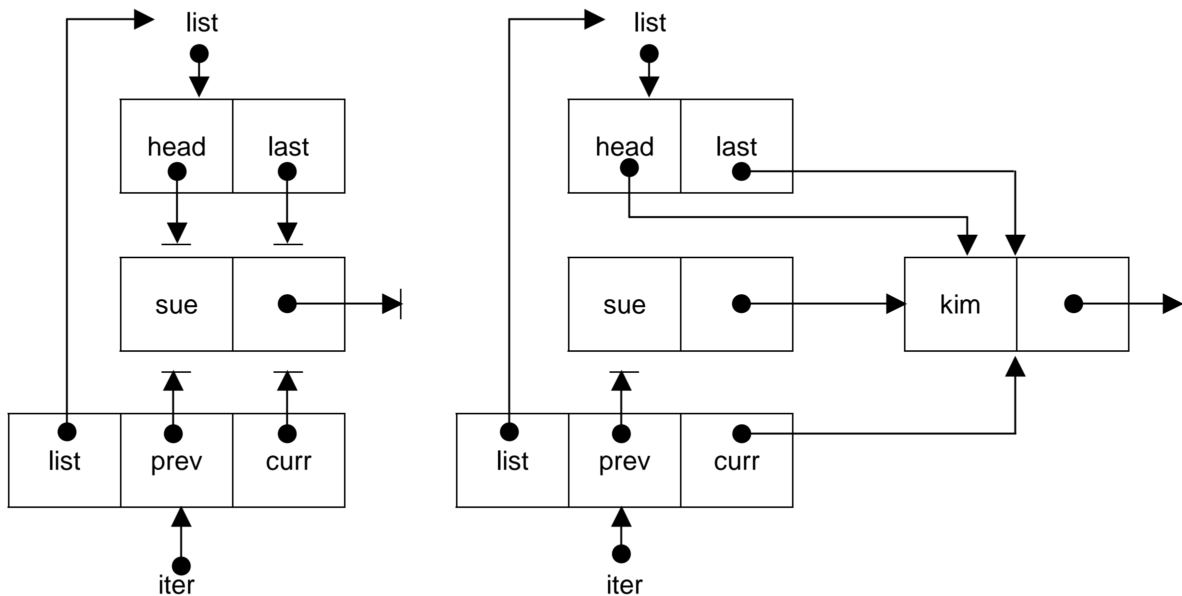
5.5.13 List Iterator Remove

listIteratorRemove() removes the item last visited by *listIteratorNext()*. *listIteratorNext()* and *listIteratorRemove()* both update the position of the list iterator, which can make removal troublesome. The important point is you must call *listIteratorNext()* before you call *litsIteratorRemove()*.

If *current* is at the *head* node, we have two possible situations: either the list contains just one node, or it contains many.

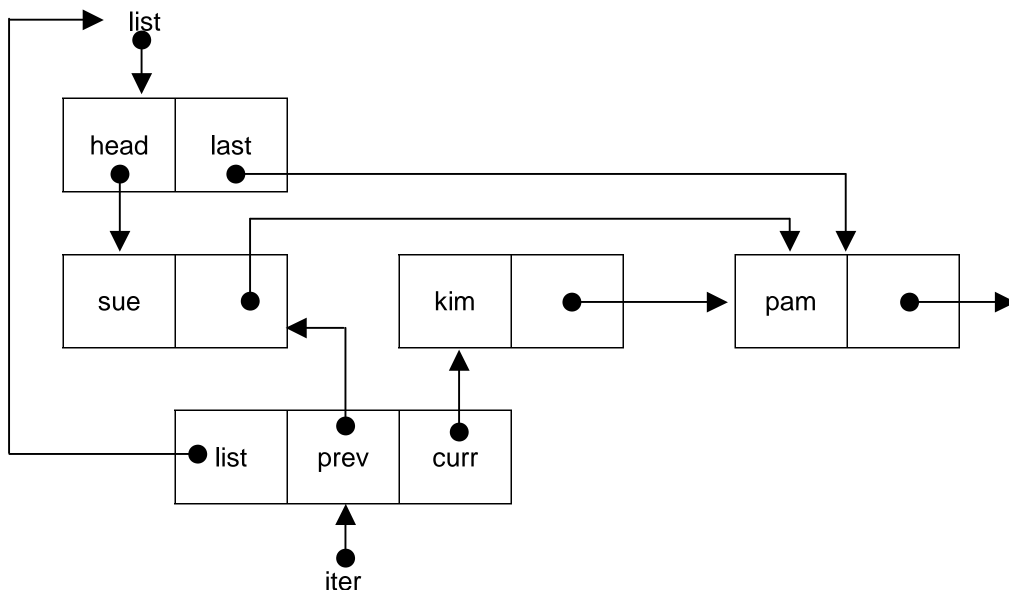


If the list contains just one node, we set *head*, *last* and *current* to *NULL*. If the list contains several nodes we set both *head* and *current* to point to the next node.




```
if (iter->current == iter->list->head) {
  if (iter->list->head == iter->list->last)
    iter->list->last = NULL;
  iter->list->head = iter->list->head->next;
  iter->current = iter->list->head;
}
```

If *current* is not at the head node we adjust the pointers to by-pass the node pointed to by *current*.




```
iter->previous->next = iter->current->next;
iter->current = iter->previous;
```

If we are discarding the last node, we need to update *list->last*.

```
if (iter->current->next == NULL)
  iter->list->last = iter->previous;
```

Having by-passed the node to be deleted, we need to discard the node with a call to *disposeNode()*. Here is the entire function.

```
/* listIteratorRemove: removes the last item retrieved by
   listIteratorNext */
List listIteratorRemove(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorRemove: list is NULL");
  if (isEmptyList(iter->list))
    listError("listIteratorRemove: list is empty");
  ListNode n = NULL;
  if (iter->current == iter->list->head) {
    n = iter->list->head;
    if (iter->list->head == iter->list->last)
      iter->list->last = NULL;
    iter->list->head = iter->list->head->next;
    iter->current = iter->list->head;
  }
  else {
    if (iter->previous == NULL)
      listError("listIteratorRemove: previous is NULL");
    n = iter->current;
    if (iter->current->next == NULL)
      iter->list->last = iter->previous;
    iter->previous->next = iter->current->next;
    iter->current = iter->previous;
  }
  iter->previous = NULL;
  disposeListNode(&n);
  return iter->list;
}
```

5.5.14 Implementation

The entire implementation of the list data structure is shown below.

```
/* list.c */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include "list.h"

typedef struct ListNodeStruct *ListNode;

struct ListNodeStruct {
  void *data;
  ListNode next;
};

struct ListHeader {
  ListNode head;
  ListNode last;
  Duplicate duplicate;
};
```

```
struct ListIteratorStruct {
  ListNode previous;
  ListNode current;
  List list;
};

/* listError: reports list errors, halt program execution */
int listError(const char *error)
{
  printf("\n%s\n", error);
  exit(1);
}

/* newList: returns a new, empty list */
List newList(Duplicate dup)
{
  List list = malloc(sizeof(struct ListHeader));
  if (list == NULL)
    listError("newList: out of memory");
  list->head = NULL;
  list->last = NULL;
  list->duplicate = dup;
  return list;
}

/* isEmptyList: returns 0 if list is empty */
int isEmptyList(const List list)
{
  if (list == NULL)
    listError("isEmptyList: list is NULL");
  return (list->head == NULL && list->last == NULL);
}

/* newListNode: returns a new, empty list node */
ListNode newListNode()
{
  ListNode n = malloc(sizeof(struct ListNodeStruct));
  if (n == NULL)
    listError("listNode: out of memory");
  n->next = NULL;
  n->data = NULL;
  return n;
}

/* addListItem: adds an item onto the last end of a list */
List addListItem(void * const data, List list)
{
  if (list == NULL)
    listError("addListItem: list is NULL");
  ListNode n = newListNode();
  n->data = list->duplicate(data);
  if (isEmptyList(list)) {
    list->head = n;
    list->last = n;
  }
  else {
    list->last->next = n;
    list->last = n;
  }
  return list;
}
```

```
/* newListIterator: returns a new list iterator. */
ListIterator newListIterator(const List list)
{
  ListIterator iter = (ListIterator)malloc(
                      sizeof(struct ListIteratorStruct));
  if (iter == NULL)
    listError("newListIterator: out of memory");
  iter->previous = NULL;
  iter->current = NULL;
  iter->list = list;
  return iter;
}

/* listIteratorHasNext: returns true if the list has an item not yet
   visited in the current iteration */
int listIteratorHasNext(const ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorHasNext: list is NULL");
  if (isEmptyList(iter->list))
    return 0;
  if (iter->current == NULL)
    return iter->list->head != NULL;
  return iter->current->next != NULL;
}

/* listIteratorNext: returns the next item in the current iteration
*/
void *listIteratorNext(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorNext: list is NULL\n");
  if (isEmptyList(iter->list))
    listError("listIteratorNext: list is empty");
  if (!listIteratorHasNext(iter))
    listError("listIteratorNext: no next node");
  iter->previous = iter->current;
  if (iter->current == NULL)
    iter->current = iter->list->head;
  else
    iter->current = iter->current->next;
  return iter->list->duplicate(iter->current->data);
}

/* listIteratorAdd: inserts item before the current iterator position
*/
List listIteratorAdd(void *data, ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorAddItem: list is NULL");
  ListNode n = newListNode();
  n->data = iter->list->duplicate(data);

 if (iter->list->head == NULL) {
    iter->list->head = n;
    iter->list->last = n;
  }
  else if (iter->current == NULL) {
    n->next = iter->list->head;
    iter->list->head = n;
  }
```

```
  else {
    n->next = iter->current;
    if (iter->list->head == iter->current)
      iter->list->head = n;
    else if (iter->previous != NULL)
      iter->previous->next = n;
  }
  iter->previous = NULL;
  return iter->list;
}

/* disposeListNode: deallocates the given node */
ListNode disposeListNode(ListNode *n)
{
  free(*n);
  *n = NULL;
  return *n;
}

/* listIteratorRemove: removes the last item retrieved by
  listIteratorNext */
List listIteratorRemove(ListIterator iter)
{
  if (iter->list == NULL)
    listError("listIteratorRemove: list is NULL");
  if (isEmptyList(iter->list))
    listError("listIteratorRemove: list is empty");
  ListNode n = NULL;
  if (iter->current == iter->list->head) {
    n = iter->list->head;
    if (iter->list->head == iter->list->last)
      iter->list->last = NULL;
    iter->list->head = iter->list->head->next;
    iter->current = iter->list->head;
  }
  else {
    if (iter->previous == NULL)
      listError("listIteratorRemove: previous is NULL");
    n = iter->current;
    if (iter->current->next == NULL)
      iter->list->last = iter->previous;
    iter->previous->next = iter->current->next;
    iter->current = iter->previous;
  }
  iter->previous = NULL;
  disposeListNode(&n);
  return iter->list;
}

/* disposeListIterator: deallocates memory occupied by ListIterator
*/
ListIterator disposeListIterator(ListIterator *iter)
{
  free(*iter);
  *iter = NULL;
  return *iter;
}
```

```
/* disposeList: deallocates memory occupied by list */
List disposeList(List *list)
{
  if (*list == NULL)
    listError("disposeList: list is NULL");

  ListNode p = (*list)->head;
  ListNode anchor = NULL;
  while (p != NULL) {
    anchor = p->next;
    disposeListNode(&p);
    p = anchor;
  }
  free(*list);
  *list = NULL;
  return *list;
}
```

Exercise 5.1

1. Implement the list data structure described above as a double-linked list, where each node has a pointer to the next node and a pointer to the previous node.

2. Implement functions to add and to remove from the head of a double-linked list.

3. Implement functions to add and remove from the last end of a double-linked list.

4. Implement list iterator functions named *listIteratorPevious* and *listIteratorHasPrevious*. *listIteratorHasPrevious()* returns true if there is another item to visit when traversing from last to head in the list. *listIteratorPrevious()* returns the previous data item.

Bibliography

Kernighan B, Ritchie D, *The C Programming language*, Prentice Hall, 1988
Mark Williams Company, *ANSI C - A lexical Guide*, Prentice Hall 1988
Stubbs D and Webre N *Data Structures with Abstract Data Types and Modula-2*, Brooks/Cole 1987
Horstmann C *www.cs.jhu.edu/~pari/600.107/Horstmann/slides/Ch19/ch19.html* accessed July 2010