Programmer's Guide to the LST Facility

A Facility for Manipulating Linked Lists

Thomas R. Leith

Mallinckrodt Institute of Radiology
Electronic Radiology Laboratory
510 South Kingshighway Boulevard
St. Louis, Missouri 63110
314/362-6965 (Voice)
314/362-6971 (FAX)

Version 2.10.0

August 3 1998

This document describes a generalized software facility for creating and maintaining linked lists. Access routines are provided to treat the lists as stacks, queues, and doubly-linked lists.

1 Introduction

The LST facility is a set of functions and type definitions that implement a doubly-linked list and provides access mechanisms to the list contents.

The LST facility allows access to a list in three ways:

- As a queue of items
- As a stack of items
- As an ordered list of items

The facility will not prevent mixing of access modes, but as a matter of style, this is not recommended. All lists are doubly-linked and have head ends and tail ends. The sections below describe how to access the items in a list.

1.1 The List as a Queue

The caller may treat a list as a queue of data items. The subset of routines appropriate to queue semantics are:

LST_Enqueue	Adds a new item to the tail end of the list.
LST_Dequeue	Removes an item from the head end of the list.
LST_Head	Examines the item at the head of the list without dequeueing it.
LST_Count	Returns the number of items that are in the queue.

The facility does not prevent using other access routines on a queue.

1.2 The List as a Stack

The caller may treat a list as a stack of data items. The subset of routines appropriate to stack semantics are:

LST_Push	Adds a new item to the head end of the list.
LST_Pop	Removes an item from the head end of the list.
LST_Top	Examines the item at the top of the stack without popping it.
LST_Count	Returns the number of items that are on the stack.

1.3 The List as an Ordered List

A list may also be treated as an ordered list of items. The program is responsible for maintaining the order of the list; the LST facility has no built-in ordering functionality.

The notion of position in the list is important to this discussion. To allow orderly list traversal, the facility provides a position context. If the program is traversing the list strictly in order, there is no real need to ``remember'' the last item accessed. The facility can retrieve the next or previous item. The next item is defined to be the adjacent item towards the tail end. The previous item is defined to be the adjacent item towards the head end of the list. Figure 1-1 illustrates a list whose current item is item B. Next is C and previous is A.

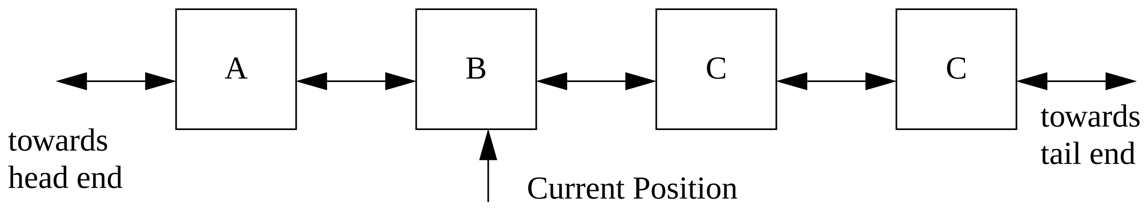

Figure 1-1. Position Example

The subset of routines that provide ordered-list semantics is:

LST_Position	Positions the pointer at an arbitrary item in the list.
LST_Next	Positions the pointer one item closer to the tail end of the list from the current position.
LST_Current	Returns a pointer to the item at the current position.
LST_Previous	Positions the pointer one item closer to the head end of the list from the current position.
LST_Insert	Inserts a new item into the list ``near'' (immediately before or after) the current position.
LST_Remove	Removes the item at the current position from the list.

Using the list illustrated in Figure 1-1 as a starting point, a call to LST_Next() would make current item C; a call to LST_Previous() would make current item A.

When LST_Insert() places a new item in the list, it puts it ``near'' the current position. Use again the list illustrated in Figure 1-1 as an example. After insertion, the newly-inserted item becomes current. When inserting a new item, the caller specifies where the new item is to be inserted. If the caller specifies LST_K_BEFORE, the new item will be inserted ``before'' the current item; that is, towards the head-end of the list. The new item will appear between items A and B. If LST_K_AFTER is specified, the new item will be inserted ``after'' the current item; that is, towards the tail-end of the list. The new item will appear between items B and C.

The LST facility also allows specification of where to move the current position after an item is removed. Once again using Figure 1-1, if the caller removes the current item from the list, and specifies LST_K_BEFORE,item A will be current after the operation is complete. Conversely, if the caller specifies LST_K_AFTER, item C will be current.The concept of current position applies only when using the list with the ordered-list semantics. It is sometimes useful, however, to traverse a queue or stack for reporting on progress, etc. The LST facility will allow this, but once again, caution is advised.

2 Data Structures

Items to be managed by the LST facility must provide space for the facility to use while managing the items. This implies that they must be specially declared in the C program. The facility requires two pointer-lengths of space at zero offset from the beginning of each item. One way to accommodate the facility is to declare structures like this:

```
typedef struct {
  void    *reserved[2];
  the rest of your declaration
}LIST_ITEM;
```

This declares an array of two pointers to void at the beginning of each list item, as the facility requires.

3 Return Values

The LST facility uses the COND facility to form and report conditions. The COND facility is documented in the Programmer's Guide to the COND facility. Some LST routines return a condition value; others return a pointer to an item. Generally, routines that add to a list return a condition, and those which access list items return a pointer to the item. There are four condition values the LST facility may return:

LST_NORMAL	Normal, successful completion.
LST_LISTNOTEMPTY	Attempt to destroy a non-empty list.
LST_BADEND	Encountered unrecognized direction value.

```
LST_NOCURRENTOperation requires an item to be current, and no
     position context is established.
```

4 Include Files

All applications that use the LST facility should include these files in the following order:

```
#include “dicom.h”
#include “condition.h”
#include “lst.h”
```

5 LST Routines

This section provides detailed documentation for each LST facility routine.

LST_Count

Name

LST_Count–return the number of items in the list.

Synopsis

```
unsigned long LST_Count(LST_HEAD **list)
```

list	The LST_HEAD returned by LST_Create()

Description

The LST facility keeps track of how many items are contained in each list. This routine returns the number of items in the list.

Return Values

The number of items in the list

Name

LST_Create –create a new list.

Synopsis

```
LST_HEAD  *LST_Create(void)
```

Description

LST_Create creates a new, empty list and returns the pointer to the head of the list to the caller. The list head is used for subsequent list operations. The function returns NULL if it is unable to create a new list. A NULL return value usually indicates a memory allocation problem.

Return Values

A list head

NULL

Name

LST_Current–return a pointer to the current list item.

Synopsis

```
LST_NODE * LST_Current(LST_HEAD **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Current returns a pointer to the current list item, but does not remove the item from the list. If the list is empty or there is no current pointer set, the function returns NULL.

Return Values

Pointer to current item

NULL

Name

LST_Dequeue–remove an item from the head-end of the list.

Synopsis

```
LST_NODE *LST_Dequeue(LST_HEAD  **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Dequeue removes an item from the head-end of a list and returns a pointer to the item. If the list is empty, the function returns NULL.

Return Values

Pointer to a LST_NODE

NULL

Name

LST_Destroy –destroy a list.

Synopsis

```
CONDITION LST_Destroy(LST_HEAD  **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Destroy destroys a list created with LST_Create().

Notes

You must explicitly remove all items from a list before it can be destroyed.

Return Values

```
LST_NORMAL
LST_LISTNOTEMPTY
```

Name

LST_Enqueue –place a new item at the tail-end of a list.

Synopsis

```
CONDITION LST_Enqueue(LST_HEAD **list, LST_NODE *item)
```

list	The LST_HEAD returned by LST_Create()
item	A pointer to the new item

Description

LST_Enqueue adds an item to the tail-end of the list.

Return Values

LST_NORMAL

Name

LST_Head –return a pointer to the item at the head-end of a list.

Synopsis

```
LST_NODE *LST_Head(LST_HEAD  **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Head returns a pointer to the item at the head-end of a list. This function does not remove the item from the list. LST_Head returns NULL if the list is empty.

Notes

This function is often used to establish a current position context when using list semantics. It is also used to define the LST_Top macro for stack semantics, and the LST_Front macro for queue semantics. LST_Top returns the top of a stack; LST_Front returns the front element of a queue.

Return Values

A pointer to a LST_NODE

NULL

Name

LST_Insert –insert an new item into a list.

Synopsis

```
CONDITION LST_Insert(LST_HEAD **list, LST_NODE *item
                    LST_END where)
```

list	The LST_HEAD returned by LST_Create()
item	A pointer to the new item to be managed
where	Where to insert the new node

Description

LST_Insert inserts an item at the current position in the list. The newly-inserted item becomes current.

Notes

The new item may be inserted adjacent to the current item, either before or after depending on the value of where. Before implies toward the “head” of the list; after implies toward the “tail” of the list. Symbolic values LST_K_BEFORE and LST_K_AFTER are defined in the header file lst.h.

Return Values

LST_NORMAL
LST_BADEND
LST_NOCURRENT

Name

LST_Next –make current the adjacent item in the direction of the tail-end of the list, and return a pointer to the item.

Synopsis

```
LST_NODE*LST_Next(LST_HEAD **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Next traverses the list. LST_Next makes current the adjacent item in the direction of the tail-end of the list and returns a pointer to that item. LST_Next returns NULL under the following conditions:

- The list is empty.
- There is no current pointer set.
- The current pointer is at the tail of the list, implying that there is no next item.

Notes

A current position must be established before calling LST_Next.

Return Values

A pointer to a LST_NODE

`NULL`

Name

LST_Pop –remove the item at the head-end of the list and return a pointer to the item.

Synopsis

```
LST_NODE*LST_Pop(LST_HEAD **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Pop pops an item off the top of a stack and returns a pointer to the item. If the list is empty, LST_Pop returns NULL.

Return Values

A pointer to a LST_NODE

`NULL`

Name

LST_Position –make current an arbitrary item in a list.

Synopsis

```
LST_NODE* LST_Position(LST_HEAD **list, LST_NODE *node)
```

list	The LST_HEAD returned by LST_Create()
node	A pointer to the node to make current

Description

LST_Position sets the current-position context to an arbitrary node in the list. It returns a pointer to that node. If the caller tries to make a node “current” which is not in the list, LST_Position returns NULL.

Notes

LST_Position must be called before LST_Next, LST_Previous, or LST_Current have any meaning. The function is often used in conjunction with LST_Head and LST_Tail to establish a context.

Return Values

`NULL`

Node

Name

LST_Previous –make current the adjacent item in the direction of the head-end of the list, and return a pointer to the item.

Synopsis

```
LST_NODE *LST_Previous(LST_HEAD **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Previous makes current the adjacent item in the direction of the head-end of the list. The function returns a pointer to the adjacent item in the direction of the head-end of the list, or NULL if there is no adjacent item, or if no current position context has been established.

Notes

A current position must be established before calling LST_Previous.

Return Values

A pointer to a LST_NODE

`NULL`

Name

LST_Push –add an item at the head-end of the list.

Synopsis

```
CONDITION LST_Push(LST_HEAD **list, LST_NODE *item)
```

list	The LST_HEAD returned by LST_Create()
item	A pointer to the item to add

Description

LST_Push pushes an item onto the head end of the list (for treating the list as a stack).

Return Values

```
LST_NORMAL
```

Name

LST_Remove –remove an item from a list

Synopsis

```
LST_NODE *LST_Remove(LST_HEAD **list,  LST_END where)
```

list	The LST_HEAD returned by LST_Create()
where	Where to move the current item pointer

Description

LST_Remove removes an item from the current position on the list and returns a pointer to the item. This function returns NULL if the list is empty or there is no current item.

Notes

After the current item is removed from the list, the new current item will be an adjacent item depends on the value of where. ``Before” implies ‘towards the head-end of the list’. ``After’’ implies `towards the tail-end of the list’. Symbolic values LST_K_BEFORE and LST_K_AFTER are defined in the header file lst.h. If there is no item there to make current, the current-position context becomes undefined, but the item is still removed from the list.

Return Values

A pointer to the item removed from the list.

`NULL`

LST_Tail

Name

LST_Tail –return a pointer to the item at the tail-end of a list

Synopsis

```
LST_NODE *LST_Tail(LST_HEAD  **list)
```

list	The LST_HEAD returned by LST_Create()

Description

LST_Tail returns a pointer to the item at the tail-end of a list. If the list is empty, LST_Tail returns NULL.

Notes

This function is often used to establish a current position context when using list semantics.

Return Values

A pointer to a LST_NODE

`NULL`

6 Code Examples

6.1 Establishing a Current-Position Context

Several LST routines require a current-position context be established before making calls. This is often done by using LST_Head() to return a pointer to the head item. An example follows.

```
list = LST_Create();

for (i = 0; i < 10; i++) {
  item = malloc(sizeof(*item));
  item->value = i;
  status = LST_Insert(&list, item, LST_K_AFTER);

}

(void)LST_Position(&list, LST_Head(&list));

for (i = 0; i < LST_Count(&list); i++) {
  item = LST_Next(&list);
  printf(“item value[%d] = %d\n”, i+1, item->value);
}
```

The program will print the item values in the order they were inserted into the queue. If the items were inserted with LST_K_BEFORE, the call to LST_Position() would be redundant, and the item values would be printed in reverse numeric order.