Subject: Re: linked list example (LONG)
Posted by davidf on Wed, 02 Sep 1998 07:00:00 GMT
View Forum Message <> Reply to Message

Doug Larson (djl@REMOVECAPSloki.srl.caltech.edu) writes:

> Here is a set of linked list modules I made for my own use. I hesitate
> to post them since they are not "bullet-proof" but they do the job.
> Hopefully someone will be inspired to write a better open source
> linked list library.

I'm not really ready to shoot my shareware idea yet, but
here is the sort of thing I had in mind to put in the
library. This is the LINKEDLIST object as it existed about
a week ago before I added the ability to interactively
edit the list. (Cut, Copy, Paste, Move Up, Move Down, etc.)

The basics are in place to implement the editing functionality
on your own, or to write your own "command editor" or
"command recorder". Having a linked list tends to generate
about 100 ideas a day for what the darn thing can be used
for, just about all of them really neat. :-)

You can learn all about linked lists and other kinds of
object programming, by the way, in courses offered by RSI
and some of their distributors. Yours truly will be
presenting the normal four day course at RSI headquarters
in Boulder September 22-25 and I will be teaching a two-day
intensive course outside of London, England for Floating Point
Systems (IDL distributors in the UK) on October 15-16.
Either Beau Leeger of RSI or I am teaching these courses
somewhere just about every month.

Cheers,

David

David Fanning, Ph.D.
Fanning Software Consulting
E-Mail: davidf@dfanning.com
Phone: 970-221-0438, Toll-Free Book Orders: 1-888-461-0155
Coyote's Guide to IDL Programming: http://www.dfanning.com/

```
;+
; NAME:
;  LINKEDLIST__DEFINE
```

```
;
; PURPOSE:
;   The purpose of this program is to implement a list that
;   is linked in both the forward and backward directions. There
;   is no restriction as to what can be stored in a linked list
;   node. The linked list is implemented as an object.
;
; AUTHOR:
;   FANNING SOFTWARE CONSULTING
;   David Fanning, Ph.D.
;   2642 Bradbury Court
;   Fort Collins, CO 80521 USA
;   Phone: 970-221-0438
;   E-mail: davidf@dfanning.com
;   Coyote's Guide to IDL Programming: http://www.dfanning.com/
;
; CATEGORY:
;  General programming.
;
; CALLING SEQUENCE:
;  mylist = Obj_New('LINKEDLIST', item)
;
; OPTIONAL INPUTS:
;   item: The first item added to the list. Items can be any
;     valid IDL variable type.
;
; COMMON BLOCKS:
;  Are you kidding?!
;
; RESTRICTIONS:
;  Be sure to destroy the LINKEDLIST object when you are finished
;   with it: Obj_Destroy, mylist
;
;   Node index numbers start at 0 and go to n-1, where n is the
;   number of items in the list.
;
; PUBLIC METHODS:
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;
;
; PRO LINKEDLIST::ADD, item, index, AFTER=after, BEFORE=before
;
;   The ADD method adds a data item to the list.
;
;   Parameters:
;
;   item: The data item to be added to the list. Required.
;
```

```
;   index: The location in the list where the data item is
;     to be added. If neither the AFTER or BEFORE keyword is
;     set, the item is added AFTER the item at the index location.
;     If index is missing, the index points to the last item in
;     the list. Optional.
;
;   Keywords:
;
;   AFTER: If this keyword is set, the item is added after the
;     item at the current index.
;
;   BEFORE: If this keyword is set, the item is added before the
;     item at the current index.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;
;
; PRO LINKEDLIST::DELETE, index, ALL=all
;
;   The DELETE method deletes an item from the list.
;
;   Parameters:
;
;   index: The location in the list where the data item is
;     to be delete. If index is missing, the index points to
;     the last item in the list. Optional.
;
;   Keywords:
;
;   ALL: If this keyword is set, all items in the list are deleted.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;
;
; FUNCTION LINKEDLIST::GET_COUNT
;
;   The GET_COUNT method returns the number of items in the list.
;
;   Return Value: The number of items stored in the linked list.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;;;;;
;
; FUNCTION LINKEDLIST::GET_ITEM, index
;
;   The GET_ITEM method returns a pointer to the specified data
;   item from the list.
;
;   Parameters:
;
;   index: The location in the list from which the data item is
```

```
;     to be retrieved. If not present, the last item in the list
;     is retrieved. Optional.
;
;   Return Value: A pointer to the specified data item stored
;     in the list.
;
;-------------------------------------------------------------------------
;
; FUNCTION LINKEDLIST::GET_NODE, index
;
;   The GET_NODE method returns a pointer to a specified node
;   from the list.
;
;   Parameters:
;
;   index: The location in the list from which the data node is
;     to be retrieved. If not present, the last node in the list
;     is retrieved. The node is a stucture with three fields:
;     Previous is a pointer to the previous node in the list.
;     Next is a pointer to the next node in the list. A null pointer
;     in the previous field indicates the first node on the list. A
;     null pointer in the next field indicates the last node on the
;     list. The item field is a pointer to the item stored in the
;     node. Optional.
;
;   Return Value: A pointer to the specified node structure
;     in the linked list.
;
;-------------------------------------------------------------------------
;
; PRO LINKEDLIST::HELP, PRINT=print
;
; The HELP method performs a HELP command on each item
; in the linked list.
;
;   Keywords:
;
;    PRINT: If this keyword is set, the PRINT command is used
;      instead of the HELP command on the items in the list.
;
;-------------------------------------------------------------------------
;
; PRO LINKEDLIST::MOVE_NODE, nodeIndex, location, BEFORE=before
;
;   The MOVE_NODE method moves a list node from one location to another.
;
;   Parameters:
;
```

```
;   nodeIndex: The location in the list of the node you are moving.
;     Required.
;
;   location: The location (index) you are moving the node to. If
;     location is missing, the location points to the node at the
;     end of the list.
;
;   Keywords:
;
;    BEFORE: If this keyword is set, the node is added to the
;      list before the location node. Otherwise, it is added after
;      the location node.
;
;-------------------------------------------------------------- ---------
;
; EXAMPLE:
;
;   mylist = Obj_New("LINKEDLIST", 5)
;   mylist->Add, 10
;   mylist->Add, 7.5, 1, /Before
;   mylist->Add, 12.5
;   mylist->Help
;   mylist->Delete
;   mylist->Help, /Print
;   Obj_Destroy, mylist
;
; MODIFICATION HISTORY:
;  Written by: David Fanning, 25 August 98.
;-


PRO LINKEDLIST::MOVE_NODE, nodeIndex, location, Before=before

; This method moves the requested node to a new location.
; The node is added AFTER the target location, unless the BEFORE
; keyword is used.

Catch, error
IF error NE 0 THEN BEGIN
   RETURN ; Silently
ENDIF

   ; A node index is required.

IF N_Elements(nodeIndex) EQ 0 THEN BEGIN
   ok = Dialog_Message('A node index is required in MOVE_NODE method.')
   RETURN
ENDIF
```

```
   ; If location is not specified the node is moved to the
   ; end of the list.

IF N_Elements(location) EQ 0 THEN BEGIN
   location = (self->Get_Count()) - 1
ENDIF

   ; Add the node to the list.

currentNode = self->Get_Node(nodeIndex)
IF Keyword_Set(before) THEN BEGIN
   self->Add, *(*currentNode).item, location, /Before
ENDIF ELSE BEGIN
   self->Add, *(*currentNode).item, location, /After
ENDELSE

   ; Delete the node from its current location.

self->Delete, nodeIndex

END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::DELETE_NODE, index

; This method deletes the indicated node from the list.

IF self.count EQ 0 THEN BEGIN
   ok = Dialog_Message('No nodes to delete.')
   RETURN
ENDIF

IF index GT (self.count - 1) THEN BEGIN
   ok = Dialog_Message('No node with the requested index number.')
   RETURN
ENDIF

   ; Get the current node and free the item pointer.

currentNode = self->Get_Node(index)
Ptr_Free, (*currentNode).item

   ; Is this the last node?

IF index EQ (self.count - 1) THEN self->Delete_Last_Node
```

```
    ; Is this the first node in the list?

IF NOT Ptr_Valid((*currentNode).previous) THEN BEGIN
    nextNode = (*currentNode).next
    Ptr_Free, (*nextNode).previous
    (*nextNode).previous = Ptr_New()
    self.head = nextNode
ENDIF ELSE BEGIN
    previousNode = (*currentNode).previous
    nextNode = (*currentNode).next
    (*nextNode).previous = previousNode
    (*previousNode).next = nextNode
ENDELSE

   ; Release the currentNode pointer.

Ptr_Free, currentNode
self.count = self.count - 1
END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::DELETE_LAST_NODE

; This method deletes the last node in the list.

IF self.count EQ 0 THEN RETURN

currentNode = self.tail
Ptr_Free, (*currentNode).item

    ; Is this the last node in the list?

IF NOT Ptr_Valid((*currentNode).previous) THEN BEGIN
    self.head = Ptr_New()
    self.tail = Ptr_New()
    self.count = 0
    Ptr_Free, (*currentNode).next
ENDIF ELSE BEGIN
    previousNode = (*currentNode).previous
    self.tail = previousNode
    Ptr_Free, (*self.tail).next
    (*self.tail).next = Ptr_New()
    self.count = self.count - 1
ENDELSE
```

```
   ; Release the currentNode pointer.

Ptr_Free, currentNode
END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::DELETE_NODES

; This method deletes all of the nodes.

WHILE Ptr_Valid(self.head) DO BEGIN
    currentNode = *self.head
    Ptr_Free, currentNode.previous
    Ptr_Free, currentNode.item
    self.head = currentNode.next
ENDWHILE

    ; Update the count.

self.count = 0

END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::DELETE, index, All=all

; This method is the public interface to the private DELETE_+ methods.
; If INDEX is not specified, the last item on the list is always deleted.
; The ALL keyword will delete all the items on the list.

   ; Delete all the nodes?

IF Keyword_Set(all) THEN BEGIN
   self->Delete_Nodes
   RETURN
ENDIF

   ; Check for index. If there is none, delete last node on list.

IF N_Elements(index) EQ 0 THEN BEGIN
   self->Delete_Last_Node
   RETURN
ENDIF
```

```
   ; Delete specified node.

self->Delete_Node, index

END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::ADD_AFTER, item, index

; This method adds an item node AFTER the item specified by
; the index number.

   ; Be sure there is an item to add.

IF N_Elements(item) EQ 0 THEN BEGIN
   ok = Dialog_Message('Must pass an ITEM to add to the list.')
   RETURN
ENDIF

   ; If no index is specified, add the item to the end of the list.

IF N_Elements(index) EQ 0 THEN BEGIN
   self->Add_To_End, item
   RETURN
ENDIF

   ; If index is equal to the number of nodes, add the item to
   ; the end of the list.

IF index EQ self.count THEN BEGIN
   self->Add_To_End, item
   RETURN
ENDIF

   ; Create a new node and store the item in it.

currentNode = Ptr_New( {LINKEDLIST_NODE} )
(*currentNode).item = Ptr_New(item)
self.count = self.count + 1

   ; Get the node currently located at the index.

indexNode = self->Get_Node(index)

   ; Get the node that follows the indexNode.
```

```
nextNode = (*indexNode).next

   ; Update pointers.

(*indexNode).next = currentNode
(*currentNode).previous = indexNode
(*nextNode).previous = currentNode
(*currentNode).next = nextNode

END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::ADD_BEFORE, item, index

; This method adds an item node BEFORE the item specified by
; the index number.

   ; Be sure there is an item to add.

IF N_Elements(item) EQ 0 THEN BEGIN
   ok = Dialog_Message('Must pass an ITEM to add to the list.')
   RETURN
ENDIF

   ; If no index is specified or the index is 0,
   ; add the item to the head of the list.

IF N_Elements(index) EQ 0 THEN index = 0

   ; Create a new node and store the item in it.

currentNode = Ptr_New( {LINKEDLIST_NODE} )
(*currentNode).item = Ptr_New(item)
self.count = self.count + 1

   ; Get the node currently located at the index.

indexNode = self->Get_Node(index)

   ; Get the node that is before the indexNode.

previousNode = (*indexNode).previous

   ; Update pointers.

(*indexNode).previous = currentNode
```

```
(*currentNode).previous = previousNode
(*currentNode).next = indexNode
IF Ptr_Valid(previousNode) THEN $
   (*previousNode).next = currentNode ELSE $
   self.head = currentNode

END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::ADD_TO_END, item

; This method adds an item to the tail of the list.

   ; Be sure you have an item to add.

IF N_Elements(item) EQ 0 THEN BEGIN
   ok = Dialog_Message('Must pass an ITEM to add to the list.')
   RETURN
ENDIF

IF self.count EQ 0 THEN BEGIN

      ; Create a new node.
   currentNode = Ptr_New({ LINKEDLIST_NODE })

      ; Add the item to the node.
   (*currentNode).item = Ptr_New(item)

      ; The head and tail point to current node.
   self.head = currentNode
   self.tail = currentNode

      ; Update the node count.
   self.count = self.count + 1

ENDIF ELSE BEGIN

      ; Create a new node.
   currentNode = Ptr_New({ LINKEDLIST_NODE })

      ; Set the next field of the previous node.
   (*self.tail).next = currentNode

      ; Add the item to the current node.
   (*currentNode).item = Ptr_New(item)
```

```
        ; Set the previous field to point to previous node.
    (*currentNode).previous = self.tail

        ; Update the tail field to point to current node.
    self.tail = currentNode

        ; Update the node count.
    self.count = self.count + 1
ENDELSE
END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::ADD, item, index, Before=before, After=after

; This method is the public interface to the private ADD_+ methods.
; If INDEX is not specified, the item is always added to the end
; of the list. If INDEX is specified, but neither the BEFORE or
; AFTER keywords are used, the item is added AFTER the INDEX specified.

   ; Must supply an item to add to the list.

IF N_Elements(item) EQ 0 THEN BEGIN
   ok = Dialog_Message('Must supply an item to add to the list.')
   RETURN
END

   ; Check for index. If there is none, add to end of list.

IF N_Elements(index) EQ 0 THEN BEGIN
   self->Add_To_End, item
   RETURN
ENDIF

   ; Are keywords set?

before = Keyword_Set(before)
after = Keyword_Set(after)

   ; No BEFORE or AFTER keywords. Add to location AFTER index.

IF (before + after) EQ 0 THEN BEGIN
   self->Add_After, item, index
   RETURN
ENDIF

   ; BEFORE keyword set.
```

```
IF before THEN BEGIN
   self->Add_Before, item, index
   RETURN
ENDIF

   ; AFTER keyword set.

IF after THEN BEGIN
   self->Add_After, item, index
   RETURN
ENDIF

END
 ;------------------------------------------------------------ -------------



FUNCTION LINKEDLIST::GET_ITEM, index

; This method returns a pointer to the information
; stored in the list. Ask for the item by number or
; order in the list (list numbers start at 0).

   ; Gets last item by default.

IF N_Params() EQ 0 THEN index = self.count - 1

    ; Make sure there are items in the list.

IF self.count EQ 0 THEN BEGIN
   ok = Dialog_Message('Nothing is currently stored in the list.')
   RETURN, Ptr_New()
ENDIF

IF index GT (self.count - 1) OR index LT 0 THEN BEGIN
   ok = Dialog_Message('Sorry. Requested node is not in list.')
   RETURN, Ptr_New()
ENDIF

    ; Start at the head of the list.

currentNode = self.head

    ; Find the item asked for by traversing the list.

FOR j=0, index-1 DO currentNode = (*currentNode).next
```

```
    ; Return the pointer to the item.

RETURN, (*currentNode).item
END
 ;------------------------------------------------------------ -------------



FUNCTION LINKEDLIST::GET_NODE, index

; This method returns a pointer to the asked-for node.
; Ask for the node by number or order in the list
; (node numbers start at 0).

   ; Gets last node by default.

IF N_Params() EQ 0 THEN index = self.count - 1

    ; Make sure there are items in the list.

IF self.count EQ 0 THEN BEGIN
   ok = Dialog_Message('Nothing is currently stored in the list.')
   RETURN, Ptr_New()
ENDIF

IF index GT (self.count - 1) OR index LT 0 THEN BEGIN
   ok = Dialog_Message('Sorry. Requested node is not in list.')
   RETURN, Ptr_New()
ENDIF

    ; Start at the head of the list.

currentNode = self.head

    ; Find the item asked for by traversing the list.

FOR j=0, index-1 DO currentNode = (*currentNode).next

    ; Return the pointer to the node.

RETURN, currentNode
END
 ;------------------------------------------------------------ -------------



FUNCTION LINKEDLIST::GET_COUNT
```

```
; This method returns the number of items in the list.

RETURN, self.count
END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::HELP, Print=print

; This method performs a HELP command on the items
; in the linked list. If the PRINT keyword is set, the
; data items are printed instead.

   ; Are there nodes to work with?

IF NOT Ptr_Valid(self.head) THEN BEGIN
   ok = Widget_Message('No nodes in Linked List.')
   RETURN
ENDIF

   ; First node.

currentNode = *self.head
IF Keyword_Set(print) THEN Print, *currentNode.item ELSE $
   Help, *currentNode.item

   ; The rest of the nodes. End of list indicated by null pointer.

WHILE currentNode.next NE Ptr_New() DO BEGIN
   nextNode = *currentNode.next
   IF Keyword_Set(print) THEN Print, *nextNode.item ELSE $
      Help, *nextNode.item
   currentNode = nextNode
ENDWHILE

END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST::CLEANUP

; This method deletes all of the nodes and cleans up
; the objects pointers.

self->Delete_Nodes
Ptr_Free, self.head
```

```
Ptr_Free, self.tail
END
 ;------------------------------------------------------------ -------------



FUNCTION LINKEDLIST::INIT, item

; Initialize the linked list. Add an item if required.

IF N_Params() EQ 0 THEN RETURN, 1
self->Add, item
RETURN, 1
END
 ;------------------------------------------------------------ -------------



PRO LINKEDLIST__DEFINE

; The implementation of a LINKEDLIST object.

   struct = { LINKEDLIST, $         ; The LINKEDLIST object.
              head:Ptr_New(), $     ; A pointer to the first node.
              tail:Ptr_New(), $     ; A pointer to the last node.
              count:0L $            ; The number of nodes in the list.
              }

   struct = { LINKEDLIST_NODE, $    ; The LINKEDLIST NODE structure.
              previous:Ptr_New(), $ ; A pointer to the previous node.
              item:Ptr_New(), $     ; A pointer to the data item.
              next:Ptr_New()  $     ; A pointer to the next node.
              }

END
 ;------------------------------------------------------------ -------------
```