

linklist

a simple single and doubly linked list package
edition 2.0.2 for `linklist` version 2.0.2
11 September 2009

Diab Jerius

Copyright © 2006 Smithsonian Institution

`linklist` is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

`linklist` is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA

Table of Contents

1	Copying	1
2	Usage	3
2.1	Overview	3
2.2	Data Encapsulation	4
2.3	Node Comparison	4
3	Library Routines	7
3.1	Singly Linked Lists	7
3.1.1	sll_count	7
3.1.2	sll_delete	7
3.1.3	sll_udelete	8
3.1.4	sll_destroy	8
3.1.5	sll_destroy_head	9
3.1.6	sll_destroy_tail	9
3.1.7	sll_destroy_dnode	10
3.1.8	sll_destroy_node	10
3.1.9	sll_detach_head_node	11
3.1.10	sll_detach_node	11
3.1.11	sll_head	12
3.1.12	sll_head_node	12
3.1.13	sll_insert	13
3.1.14	sll_insert_head	13
3.1.15	sll_insert_head_node	14
3.1.16	sll_insert_node	14
3.1.17	sll_insert_tail	15
3.1.18	sll_insert_tail_node	15
3.1.19	sll_join	16
3.1.20	sll_new	17
3.1.21	sll_next	17
3.1.22	sll_next_node	18
3.1.23	sll_node_get_data	18
3.1.24	sll_node_put_data	18
3.1.25	sll_prev	19
3.1.26	sll_prev_node	20
3.1.27	sll_search	20
3.1.28	sll_search_node	21
3.1.29	sll_sizeof_node	22
3.1.30	sll_tail	22
3.1.31	sll_tail_node	22
3.1.32	sll_traverse	23
3.1.33	sll_traverse_d	23

3.1.34	sll_utraverse	24
3.1.35	sll_utraverse_d	25
3.2	Doubly Linked Lists	27
3.2.1	dll_count	27
3.2.2	dll_delete	27
3.2.3	dll_udelete	27
3.2.4	dll_destroy	28
3.2.5	dll_destroy_head	29
3.2.6	dll_destroy_tail	29
3.2.7	dll_destroy_dnode	30
3.2.8	dll_destroy_node	30
3.2.9	dll_detach_head_node	31
3.2.10	dll_detach_node	31
3.2.11	dll_head	32
3.2.12	dll_head_node	32
3.2.13	dll_insert	32
3.2.14	dll_insert_head	33
3.2.15	dll_insert_head_node	34
3.2.16	dll_insert_node	34
3.2.17	dll_insert_tail	35
3.2.18	dll_insert_tail_node	35
3.2.19	dll_join	36
3.2.20	dll_new	36
3.2.21	dll_next	37
3.2.22	dll_next_node	37
3.2.23	dll_node_get_data	38
3.2.24	dll_node_put_data	38
3.2.25	dll_prev	39
3.2.26	dll_prev_node	39
3.2.27	dll_search	40
3.2.28	dll_search_node	41
3.2.29	dll_sizeof_node	41
3.2.30	dll_tail	42
3.2.31	dll_tail_node	42
3.2.32	dll_traverse	42
3.2.33	dll_traverse_d	43
3.2.34	dll_utraverse	44
3.2.35	dll_utraverse_d	45

1 Copying

The software described by this manual is copyright © 2006 Smithsonian Institution. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

2 Usage

`linklist` is a package of routines which implements both singly and doubly linked lists.

Functions supported by this package include searches; list traversals in with actions performed at each node; detachment and reattachment of nodes.

2.1 Overview

Functions are available for both singly and doubly linked lists. Certain functions are extremely expensive for singly linked list, but are included for completeness. Those functions which operate on singly linked lists have a prefix of `'sll_'`, those which operate on doubly linked lists one of `'dll_'`. In the discussion which follows, the prefix is specified as `'?ll'` except in cases where the function is available only for a specific type of list.

This package provides two, parallel, interfaces to lists. The first, preferred, method, is to return results from searches, etc. as node data rather than as references to nodes. This helps shield the user from the vagaries of the list implementation, and, in most cases, is really what the user wants.

The second method deals directly with node handles. This is generally more efficient, but requires more care by the user. It also permits more specialized manipulations, such as moving nodes between lists without the need to create and destroy nodes. It's also the only way to have the user step through the list (there is a method of automatic list traversal with a callback function at each node). The routines which work with node handles generally have the string `'node'` in their names.

Lists are constructed with `?ll_new` and are destroyed with `?ll_delete` or `?ll_udelete`. While it is possible to insert data without any intrinsic order (inserting at the head or tail of the list, for example), it is also possible to insert data according to a user defined collating sequence. A comparison function to implement this order is passed to the list creation routines. Note that there is no consistency check made if the user chooses to mix sorted inserts with other inserts.

Nodes are created and inserted into lists with `?ll_insert`, `?ll_insert_head`, `?ll_insert_tail`. Detached nodes (detached with `?ll_detach_node`) may be inserted with `?ll_insert_node`, `?ll_insert_head_node`, and `?ll_insert_tail_node`.

Nodes can be removed from lists either by completely destroying them, via `?ll_destroy`, `?ll_destroy_head`, `?ll_destroy_tail`, `?ll_destroy_node`, or by detaching them from the list via `?ll_detach_node`. The latter is useful if a node is to be moved from one list to another. Detached nodes are themselves destroyed with `?ll_destroy_dnode`. Two lists may be joined efficiently via `?ll_join`.

A list is searched with `?ll_search`, or `?ll_search_node`. The first returns a pointer to the user data, the last a pointer to the node.

The head and tail nodes are accessed via `?ll_head`, `?ll_tail`, `?ll_head_node`, and `?ll_tail_node`. The latter two return handles to the appropriate node.

The list can be traversed and an action performed at each node traversals via `?ll_traverse` and `?ll_utraverse`. The latter provides for the passing of extra data to the user

provided action routine. Doubly linked lists can be traversed either from head to tail or from tail to head.

Once a specific node has been identified via any of the node handle retrieval routines, various manipulations are possible. It can be detached from the list (`?ll_detach_node`); you can find its predecessor or successor nodes (`?ll_next_node`, `?ll_prev_node`), predecessor or successor data (`?ll_next`, `?ll_prev`); its data can be retrieved and modified (`?ll_node_get_data`; and `?ll_node_put_data`); and you can delete it (`?ll_destroy_node`);.

The number of nodes in a list is available via `?ll_count`. The size of a node is available via `?ll_sizeof_node`.

2.2 Data Encapsulation

Each node contains a *data pointer* which associates a separate user-supplied data structure with the node. The pointer is stored at node creation, and is passed back after searches, etc. Nodes do not contain *any* user data. The user is responsible for deallocating any data when individual nodes are destroyed. When deleting entire list via `?ll_delete`, a user supplied routine will be invoked on each node's data pointer, if requested.

2.3 Node Comparison

There are two situations in which nodes will be compared, either to other nodes or to key data. During collated node insertion, the inserted node's data is compared against other nodes' data to determine the proper ordering of the nodes. This operation uses the comparison routine passed to `?ll_new` when the list is created. The second situation is during searches of the list, where node data is compared to some user specified data. Since the comparison routine passed to `?ll_new` assumes the same form for both pieces of data passed to it for comparison, using it would require that the user create a dummy user node data structure (the same as the one associated with each node), and fill it with the key data, which in most instances will probably be one field. `linklist` provides the possibility of using another comparison routine (passed to the search or destroy routines), which may compare data with two dissimilar forms. For example, assume that the user node data has the following structure, and keys on the 'id' value:

```
typedef struct
{
    int id;
    char *name;
} UserNodeData;
```

The node comparison routine, used to compare nodes during insertion, would look like this:

```
int node_node_compare(const void *dp1, const void *dp2)
{
    return ((UserNodeData *)dp1)->id - ((UserNodeData *)dp2)->id;
}
```

If you want to search the resultant list, and not create a dummy `UserNodeData` structure, construct the search/destroy comparison routine as follows:


```
int key_node_compare(const void *dp1, const void *dp2)
{
    return *((int *) dp1) - ((UserData *)dp2)->id;
}
```

and pass the search/destroy routine a pointer to an `int` set equal to the id for which you're searching.

3 Library Routines

3.1 Singly Linked Lists

3.1.1 sll_count

Determine the number of nodes in a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

size_t sll_count(SLinkList ull);
```

Parameters

```
SLinkList ull
    the list to count
```

Description

This function returns the number of nodes in a list. It is an inexpensive routine to call.

Returns

It returns the number of nodes in the list, '0' if the passed pointer is `NULL`.

3.1.2 sll_delete

Delete a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

void sll_delete(
    SLinkList ull,
    void (*ufree)(void *)
);
```

Parameters

```
SLinkList ull
    the linked list to be deleted

void (*ufree)(void *)
    a function called at each link to delete user data. it may be SLL_
    NULL_DELETE
```

Description

`sll_delete` traverses a linked list, calling a user-supplied function at each node in the list, then deleting the node. Finally it deletes the list header.

3.1.3 sll_udelete

Delete a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

void sll_udelete(
    SLinkList ull,
    void (*ufree)(void *,void *),
    void *udata
);
```

Parameters

```
SLinkList ull
    the linked list to be deleted

void (*ufree)(void *,void *)
    a function called at each link to delete user data. it may be SLL_
    NULL_DELETE

void *udata
    a pointer to data to be passed to the action routine
```

Description

`sll_udelete` traverses a linked list, calling a user-supplied function at each node in the list, then deleting the node. Finally it deletes the list header. This routine differs from `sll_delete` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

3.1.4 sll_destroy

Destroy a node in a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

void *sll_destroy(
    SLinkList ull,
    void *data,
    int (*cmp)(const void *,const void *)
);
```

Parameters

```
SLinkList ull
    the list to search

void *data
    the data to search for
```

```
int (*cmp)(const void *,const void *)
```

The address of a comparison function. Set to `SLL_NULL_CMP` to use the list's initial comparison function.

Description

`sll_destroy` searches a linked list for a node which compares equivalently to the passed data. It removes the node from the list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

It uses the passed comparison routine, if available. If not, it uses that with which the list was initialized. If *that* is not available, a segmentation violation is unavoidable. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return `-1`, `0`, or `1` if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns the node's data pointer if the node was found, `NULL` otherwise.

3.1.5 `sll_destroy_head`

Remove the head node from a singly linked list and destroy it.

Synopsis

```
#include <linklist/linklist.h>
```

```
void *sll_destroy_head(SLinkList ull);
```

Parameters

```
SLinkList ull
```

the list from which to remove the node

Description

This routine removes the head node from the specified list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer. If the list is empty, it returns `NULL`.

3.1.6 `sll_destroy_tail`

Remove the tail node from a singly linked list and destroy it.

Synopsis

```
#include <linklist/linklist.h>
```

```
void *sll_destroy_tail(SLinkList ull);
```

Parameters

SLinkedList ull
the list from which to remove the node

Description

This routine removes the tail node from the specified list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer. If the list is empty, it returns NULL.

3.1.7 sll_destroy_dnode

Deallocate the memory associated with a detached node.

Synopsis

```
#include <linklist/linklist.h>

void sll_destroy_dnode(SLLNode unode);
```

Parameters

SLLNode unode
the node to destroy

Description

This routine deallocates the memory associated with a detached node. The user must have already destroyed the data associated with the node. The node handle must have been returned by `sll_detach_node`.

3.1.8 sll_destroy_node

Remove a node from a singly linked list and destroy it.

Synopsis

```
#include <linklist/linklist.h>

void *sll_destroy_node(
    SLinkedList ull,
    SLLNode unode
);
```

Parameters

SLinkedList ull
the list from which to remove the node

SLLNode unode
the node to remove

Description

This routine removes the passed node from the specified list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer if the node is not `NULL`, `NULL` otherwise.

3.1.9 `sll_detach_head_node`

Detach the head node of a list.

Synopsis

```
#include <linklist/linklist.h>

SLLNode sll_detach_head_node(SLinkList ull);
```

Parameters

```
SLinkList ull
    the list in question
```

Description

This routine detaches the head node of a list from the list and returns a handle to the detached node.

Returns

It returns a handle to the node upon success, `NULL` on failure.

3.1.10 `sll_detach_node`

Remove a node from a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

void sll_detach_node(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

```
SLinkList ull
    the list from which to detach the node

SLLNode unode
    the node to detach
```

Description

This routine removes a node from a list without destroying it. The user is responsible for its subsequent care.

Returns

It returns a handle for the detached node.

3.1.11 sll_head

Get the data of the head node of a list.

Synopsis

```
#include <linklist/linklist.h>

void *sll_head(SLinkList ull);
```

Parameters

```
SLinkList ull
    the list in question
```

Description

Get the data of the head node of a list.

Returns

It returns the data of the head node of a list, `NULL` if the list is empty.

3.1.12 sll_head_node

Get a node handle to the head node of a list.

Synopsis

```
#include <linklist/linklist.h>

SLLNode sll_head_node(SLinkList ull);
```

Parameters

```
SLinkList ull
    the list in question
```

Description

Get a node handle to the head node of a list.

Returns

It returns the node handle of the head node of a list, `NULL` if the list is empty.

3.1.13 sll_insert

Create and insert a node into a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

int sll_insert(
    SLinkList ull,
    void *data
);
```

Parameters

```
SLinkList ull
    a handle to the list into which to insert the node

void *data
    a pointer to the new node's data
```

Description

`sll_insert` creates a node, stores the passed data pointer in it, and inserts the node in the list in the collating order determined by the comparison function with which the list was initialized by `sll_new`.

Returns

It returns zero if the insert was successful, non-zero if it was unable to create the new node

3.1.14 sll_insert_head

Create and insert a node at the head of a singly. linked list

Synopsis

```
#include <linklist/linklist.h>

int sll_insert_head(
    SLinkList ull,
    void *data
);
```

Parameters

```
SLinkList ull
    a handle to the list into which to insert the node

void *data
    a pointer to the new node's data
```

Description

This routine creates a node, inserts the passed data pointer in it, and attaches the node to the head of the list.

Returns

It returns zero if the insert was successful, non-zero if it was unable to create the new node.

3.1.15 sll_insert_head_node

Insert a detached node at the head of a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

int sll_insert_head_node(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

SLinkList ull	a handle to the list into which to insert the node
SLLNode unode	the node to insert

Description

This routine inserts a node at the head of a list. The node handle must have been obtained from `sll_detach_node`.

Returns

It returns non-zero if the node is `NULL`, zero otherwise.

3.1.16 sll_insert_node

Insert a detached node into a list.

Synopsis

```
#include <linklist/linklist.h>

int sll_insert_node(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

`SLinkList ull`
a handle to the list into which to insert the node

`SLLNode unode`
the node to insert

Description

This routine inserts a detached node into a list in the collating sequence determined by the comparison function with which the list was initialized by `sll_new`. It assumes that the list's insert/delete comparison function can be applied to the data in the passed node. The node handle must have been obtained from `sll_detach_node`.

Returns

It returns non-zero if the node is `NULL`, zero otherwise.

3.1.17 `sll_insert_tail`

Create and insert a node at the tail of a singly linked list.

Synopsis

```
#include <linklist/linklist.h>

int sll_insert_tail(
    SLinkList ull,
    void *data
);
```

Parameters

`SLinkList ull`
a handle to the list into which to insert the node

`void *data`
a pointer to the new node's data

Description

This routine creates a node, inserts the passed data pointer in it, and attaches the node to the tail of the list.

Returns

It returns zero if the insert was successful, non-zero if it was unable to create the new node.

3.1.18 `sll_insert_tail_node`

Insert a detached node at the tail of a list.

Synopsis

```
#include <linklist/linklist.h>

int sll_insert_tail_node(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

`SLinkList ull`
a handle to the list into which to insert the node

`SLLNode unode`
the node to insert

Description

This routine inserts a node at the tail of a list. The node handle must have been obtained from `sll_detach_node`.

Returns

It returns non-zero if the node is `NULL`, zero otherwise.

3.1.19 sll_join

Join two lists.

Synopsis

```
#include <linklist/linklist.h>

void sll_join(
    SLinkList dst_ull,
    SLinkList src_ull
);
```

Parameters

`SLinkList dst_ull`
the destination list

`SLinkList src_ull`
the source list

Description

This routine moves the nodes in a list to another. If the destination list has a preferred order (if a comparison function was specified when the list was created), the new nodes are inserted in order. (This implies that the data in the source list have the same format as those in the destination list.) If it has no preferred order, the source list is simply appended to the destination list. The source list is *not* destroyed, it is simply emptied.

3.1.20 `sll_new`

Create a new singly linked list.

Synopsis

```
#include <linklist/linklist.h>

SLinkList sll_new(int (*cmp)(const void *,const void *));
```

Parameters

```
int (*cmp)(const void *,const void *)
    a comparison function for use in searches of the linked list, may be
    SLL_NULL_CMP for unordered lists
```

Description

This routine creates a new list structure. If the list is to have some intrinsic order, a function defining that order should be passed. The comparison routine is called with two node data pointers as the arguments. It must return ‘-1’, ‘0’, or ‘1’ if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns a pointer to a new linked list, or NULL if it can’t allocate it

3.1.21 `sll_next`

Retrieve the data in the node following a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

void *sll_next(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

```
SLinkList ull
    the list which contains the node

SLLNode unode
    the preceding node
```

Description

Retrieve the data in the node following a given node in a list.

Returns

It returns NULL if there are no more nodes, else a pointer to the data.

3.1.22 `sll_next_node`

Get a handle to the node following a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

SLLNode sll_next_node(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

`SLinkList ull`
the list which contains the node

`SLLNode unode`
the preceding node

Description

Get a handle to the node following a given node in a list.

Returns

It returns NULL if there are no more nodes, else a handle to the node.

3.1.23 `sll_node_get_data`

Retrieve the data from a specified linked list node.

Synopsis

```
#include <linklist/linklist.h>

void *sll_node_get_data(SLLNode unode);
```

Parameters

`SLLNode unode`
the node in question

Description

Retrieve the data from a specified linked list node.

Returns

It returns the node's data pointer.

3.1.24 `sll_node_put_data`

Replace the data pointer in a given node with another.

Synopsis

```
#include <linklist/linklist.h>

void sll_node_put_data(
    SLLNode unode,
    void *data
);
```

Parameters

SLLNode unode
the node in question

void *data
the new data

Description

This function allows the calling routine to change the data that a node points to. No check is made to insure that any existing ordering of the data is maintained.

3.1.25 sll_prev

Retrieve the data in the node preceding a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

void *sll_prev(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

SLinkList ull
the list which contains the node

SLLNode unode
the following node

Description

Search a singly linked list starting at the list head and determine the node previous to the passed node. This is a very expensive operation! Before using this, think about using a doubly linked list!

Returns

It returns NULL if there is none, else a pointer to the previous node's data.

3.1.26 sll_prev_node

Get a handle to the node previous to a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

SLLNode sll_prev_node(
    SLinkList ull,
    SLLNode unode
);
```

Parameters

```
SLinkList ull
    the list which contains the node

SLLNode unode
    the following node
```

Description

Search a singly linked list starting at the list head and determine the node previous to the passed node. This is a very expensive operation! Before using this, think about using a doubly linked list!

Returns

It returns NULL if there is none, else a handle to the node.

3.1.27 sll_search

Search a linked list for a node with equivalent data.

Synopsis

```
#include <linklist/linklist.h>

void *sll_search(
    SLinkList ull,
    void *data,
    int (*cmp)(const void *,const void *)
);
```

Parameters

```
SLinkList ull
    the list to search

void *data
    the data to search for
```



```
int (*cmp)(const void *,const void *)
```

The address of a comparison function. Set to `SLL_NULL_CMP` to use the list's initial comparison function.

Description

This searches the list for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the list was initialized. If *that* is not available, a segmentation violation is unavoidable. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '-1', '0', or '1' if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns the node's data pointer if the node was found, `NULL` otherwise.

3.1.28 sll_search_node

Search a linked list for a node with equivalent data.

Synopsis

```
#include <linklist/linklist.h>

SLLNode sll_search_node(
    SLinkList ull,
    void *data,
    int (*cmp)(const void *,const void *)
);
```

Parameters

```
SLinkList ull
    the list to search
```

```
void *data
    the data to search for
```

```
int (*cmp)(const void *,const void *)
    The address of a comparison function. Set to SLL_NULL_CMP to use
    the list's initial comparison function.
```

Description

This searches the list for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the list was initialized. If *that* is not available, a segmentation violation is unavoidable. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '-1', '0', or '1' if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns a handle to the matching mode, or NULL if not found.

3.1.29 sll_sizeof_node

Get the size of a node.

Synopsis

```
#include <linklist/linklist.h>
```

```
size_t sll_sizeof_node(void);
```

Description

This routine returns the size of the internal structure used for each node. This does not include user supplied data. Please note that this is a function call, not a macro!

3.1.30 sll_tail

Get the data of the tail node of a list.

Synopsis

```
#include <linklist/linklist.h>
```

```
void *sll_tail(SLinkList ull);
```

Parameters

```
SLinkList ull  
    the list in question
```

Description

Get the data of the tail node of a list.

Returns

It returns the data of the tail node of a list, NULL if the list is empty.

3.1.31 sll_tail_node

Get a node handle to the tail node of a list.

Synopsis

```
#include <linklist/linklist.h>
```

```
SLLNode sll_tail_node(SLinkList ull);
```

Parameters

SLinkedList ull
the list in question

Description

Get a node handle to the tail node of a list.

Returns

It returns the node handle of the tail node of a list, NULL if the list is empty.

3.1.32 sll_traverse

Traverse a singly linked list, processing each node.

Synopsis

```
#include <linklist/linklist.h>

int sll_traverse(
    SLinkedList ull,
    int (*action)(void *)
);
```

Parameters

SLinkedList ull
A handle to the list to traverse

int (*action)(void *)
The user supplied action routine to be applied to each node

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer. If an invocation of the action routine returns non-zero, the traversal is aborted and the action routine's return value is returned.

Compare this to [Section 3.1.33 \[sll_traverse_d\]](#), page 23, [Section 3.1.34 \[sll_utrace\]](#), page 24, and [Section 3.1.35 \[sll_utrace_d\]](#), page 25.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.1.33 sll_traverse_d

Traverse a singly linked list, processing each node.

Synopsis

```
#include <linklist/linklist.h>
```

```
int sll_traverse_d(
    SLinkList ull,
    int (*action)(void *),
    void *ndata
);
```

Parameters

SLinkList ull
A handle to the list to traverse

int (*action)(void *)
The user supplied action routine to be applied to each node

void *ndata
a pointer to a **void *** variable which will receive the data pointer of the node which caused the traversal to be aborted. The type really is **void ****.

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer. If the action routine returns non-zero, the traversal is aborted, the current node's data pointer is stored in the location specified by the **ndata** argument, and the action routine's return value is returned.

Compare this to [Section 3.1.32 \[sll_traverse\]](#), page 23, [Section 3.1.34 \[sll_utrace\]](#), page 24, and [Section 3.1.35 \[sll_utrace_d\]](#), page 25.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.1.34 sll_utrace

Traverse a singly linked list, processing each node.

Synopsis

```
#include <linklist/linklist.h>

int sll_utrace(
    SLinkList ull,
    int (*action)(void *node,void *udata),
    void *udata
);
```

Parameters

SLinkList ull
A handle to the list to traverse

```
int (*action)(void *node,void *udata)
    The user supplied action routine applied to each node

void *udata
    a pointer to data to be passed to the action routine
```

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer as well as a pointer provided by the calling routine, thus allowing arbitrary data to be available to the action routine. If an invocation of the action routine returns non-zero the traversal is aborted and the action routine's return value is returned.

Compare this to [Section 3.1.32 \[sll-traverse\]](#), page 23, [Section 3.1.33 \[sll-traverse-d\]](#), page 23, and [Section 3.1.35 \[sll-utrace-d\]](#), page 25.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.1.35 sll-utrace-d

Traverse a singly linked list, processing each node.

Synopsis

```
#include <linklist/linklist.h>

int sll-utrace-d(
    SLinkList ull,
    int (*action)(void *node,void *udata),
    void *udata,
    void *ndata
);
```

Parameters

```
SLinkList ull
    A handle to the list to traverse

int (*action)(void *node,void *udata)
    The user supplied action routine applied to each node

void *udata
    a pointer to data to be passed to the action routine

void *ndata
    a pointer to a void * variable which will receive the data pointer of
    the node which caused the traversal to be aborted. The type really
    is void **.
```

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer as well as a pointer provided by the calling routine, thus allowing arbitrary data to be available to the action routine. If an invocation of the action routine returns non-zero, the traversal is aborted, the current node's data pointer is stored in the location specified by the `ndata` argument, and the action routine's return value is returned.

Compare this to [Section 3.1.32 \[sll_traverse\]](#), page 23, [Section 3.1.33 \[sll_traverse_d\]](#), page 23, and [Section 3.1.34 \[sll_ustraverse\]](#), page 24.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.2 Doubly Linked Lists

3.2.1 `dll_count`

Determine the number of nodes in a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

size_t dll_count(DLinkList ull);
```

Parameters

DLinkList ull
the list to count

Description

This function returns the number of nodes in a list. It is an inexpensive routine to call.

Returns

It returns the number of nodes in the list, '0' if the passed pointer is NULL.

3.2.2 `dll_delete`

Delete a doubly list.

Synopsis

```
#include <linklist/linklist.h>

void dll_delete(
    DLinkList ull,
    void (*ufree)(void *)
);
```

Parameters

DLinkList ull
the linked list to be deleted

void (*ufree)(void *)
a function called at each link to delete user data. it may be DLL_ NULL_DELETE

Description

`dll_delete` traverses a linked list, calling a user-supplied function at each node in the list, then deleting the node. Finally it deletes the list header.

3.2.3 `dll_udelete`

Delete a doubly list.

Synopsis

```
#include <linklist/linklist.h>

void dll_udelete(
    DLinkedList ull,
    void (*ufree)(void *,void *),
    void *udata
);
```

Parameters

```
DLinkedList ull
    the linked list to be deleted

void (*ufree)(void *,void *)
    a function called at each link to delete user data. it may be DLL_
    NULL_DELETE

void *udata
    a pointer to data to be passed to the action routine
```

Description

`dll_udelete` traverses a linked list, calling a user-supplied function at each node in the list, then deleting the node. Finally it deletes the list header. This routine differs from `dll_delete` in that it can pass along a pointer provided by the calling routine to the action routine, allowing arbitrary data to be available to the action routine.

3.2.4 `dll_destroy`

Destroy a node in a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

void *dll_destroy(
    DLinkedList ull,
    void *data,
    int (*cmp)(const void *,const void *)
);
```

Parameters

```
DLinkedList ull
    the list to search

void *data
    the data to search for

int (*cmp)(const void *,const void *)
    The address of a comparison function. Set to DLL_NULL_CMP to use
    the list's initial comparison function.
```


Description

`dll_destroy` searches a linked list for a node which compares equivalently to the passed data. It removes the node from the list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

It uses the passed comparison routine, if available. If not, it uses that with which the list was initialized. If *that* is not available, a segmentation violation is unavoidable. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return '-1', '0', or '1' if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns the node's data pointer if the node was found, `NULL` otherwise.

3.2.5 `dll_destroy_head`

Remove the head node from a doubly linked list and destroy it.

Synopsis

```
#include <linklist/linklist.h>

void *dll_destroy_head(DLinkedList ull);
```

Parameters

```
DLinkedList ull
    the list from which to remove the node
```

Description

This routine removes the head node from the specified list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer. If the list is empty, it returns `NULL`.

3.2.6 `dll_destroy_tail`

Remove the tail node from a doubly linked list and destroy it.

Synopsis

```
#include <linklist/linklist.h>

void *dll_destroy_tail(DLinkedList ull);
```

Parameters

```
DLinkedList ull
    the list from which to remove the node
```

Description

This routine removes the tail node from the specified list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer. If the list is empty, it returns NULL.

3.2.7 `dll_destroy_dnode`

Deallocate the memory associated with a detached node.

Synopsis

```
#include <linklist/linklist.h>

void dll_destroy_dnode(DLLNode unode);
```

Parameters

DLLNode unode
the node to destroy

Description

This routine deallocates the memory associated with a detached node. The user must have already destroyed the data associated with the node. The node handle must have been returned by `dll_detach_node`.

3.2.8 `dll_destroy_node`

Remove a node from a doubly linked list and destroy it.

Synopsis

```
#include <linklist/linklist.h>

void *dll_destroy_node(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

DLinkedList ull
the list from which to remove the node

DLLNode unode
the node to remove

Description

This routine removes the passed node from the specified list and deallocates the memory associated with it. The user must destroy the data referenced by the node's data pointer.

Returns

It returns the node's data pointer if the node is not `NULL`, `NULL` otherwise.

3.2.9 `dll_detach_head_node`

Detach the head node of a list.

Synopsis

```
#include <linklist/linklist.h>

DLLNode dll_detach_head_node(DLinkedList ull);
```

Parameters

```
DLinkedList ull
    the list in question
```

Description

This routine detaches the head node of a list from the list and returns a handle to the detached node.

Returns

It returns a handle to the node upon success, `NULL` on failure.

3.2.10 `dll_detach_node`

Remove a node from a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

void dll_detach_node(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

```
DLinkedList ull
    the list from which to detach the node

DLLNode unode
    the node to detach
```

Description

This routine removes a node from a list without destroying it. The user is responsible for its subsequent care.

Returns

It returns a handle for the detached node.

3.2.11 dll_head

Get the data of the head node of a list.

Synopsis

```
#include <linklist/linklist.h>

void *dll_head(DLinkList ull);
```

Parameters

```
DLinkList ull
           the list in question
```

Description

Get the data of the head node of a list.

Returns

It returns the data of the head node of a list, NULL if the list is empty.

3.2.12 dll_head_node

Get a node handle to the head node of a list.

Synopsis

```
#include <linklist/linklist.h>

DLLNode dll_head_node(DLinkList ull);
```

Parameters

```
DLinkList ull
           the list in question
```

Description

Get a node handle to the head node of a list.

Returns

It returns the node handle of the head node of a list, NULL if the list is empty.

3.2.13 dll_insert

Create and insert a node into a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

int dll_insert(
    DLinkedList ull,
    void *data
);
```

Parameters

DLinkedList ull
a handle to the list into which to insert the node

void *data
a pointer to the new node's data

Description

`dll_insert` creates a node, stores the passed data pointer in it, and inserts the node in the list in the collating order determined by the comparison function with which the list was initialized by `dll_new`.

Returns

It returns zero if the insert was successful, non-zero if it was unable to create the new node.

3.2.14 `dll_insert_head`

Create and insert a node at the head of a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

int dll_insert_head(
    DLinkedList ull,
    void *data
);
```

Parameters

DLinkedList ull
a handle to the list into which to insert the node

void *data
a pointer to the new node's data

Description

This routine creates a node, inserts the passed data pointer in it, and attaches the node to the head of the list.

Returns

It returns zero if the insert was successful, non-zero if it was unable to create the new node.

3.2.15 `dll_insert_head_node`

Insert a detached node at the head of a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

int dll_insert_head_node(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

`DLinkedList ull`
a handle to the list into which to insert the node

`DLLNode unode`
the node to insert

Description

This routine inserts a node at the head of a list. The node handle must have been obtained from `dll_detach_node`.

Returns

It returns non-zero if the node is `NULL`, zero otherwise.

3.2.16 `dll_insert_node`

Insert a detached node into a list.

Synopsis

```
#include <linklist/linklist.h>

int dll_insert_node(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

`DLinkedList ull`
a handle to the list into which to insert the node

`DLLNode unode`
the node to insert

Description

This routine inserts a detached node into a list in the collating sequence determined by the comparison function with which the list was initialized by `dll_new`. It assumes that the list's insert/delete comparison function can be applied to the data in the passed node. The node handle must have been obtained from `dll_detach_node`.

Returns

It returns non-zero if the node is `NULL`, zero otherwise.

3.2.17 `dll_insert_tail`

Create and insert a node at the tail of a doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

int dll_insert_tail(
    DLinkedList ull,
    void *data
);
```

Parameters

```
DLinkedList ull
    a handle to the list into which to insert the node

void *data
    a pointer to the new node's data
```

Description

This routine creates a node, inserts the passed data pointer in it, and attaches the node to the tail of the list.

Returns

It returns zero if the insert was successful, non-zero if it was unable to create the new node.

3.2.18 `dll_insert_tail_node`

Insert a detached node at the tail of a list.

Synopsis

```
#include <linklist/linklist.h>

int dll_insert_tail_node(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

`DLinkedList ull`
a handle to the list into which to insert the node

`DLLNode unode`
the node to insert

Description

This routine inserts a node at the tail of a list. The node handle must have been obtained from `dll_detach_node`.

Returns

It returns non-zero if the node is `NULL`, zero otherwise.

3.2.19 `dll_join`

Join two lists.

Synopsis

```
#include <linklist/linklist.h>

void dll_join(
    DLinkedList dst_ull,
    DLinkedList src_ull
);
```

Parameters

`DLinkedList dst_ull`
the destination list

`DLinkedList src_ull`
the source list

Description

This routine moves the nodes in a list to another. If the destination list has a preferred order (if a comparison function was specified when the list was created), the new nodes are inserted in order. (This implies that the data in the source list have the same format as those in the destination list.) If it has no preferred order, the source list is simply appended to the destination list. The source list is *not* destroyed, it is simply emptied.

3.2.20 `dll_new`

Create a new doubly linked list.

Synopsis

```
#include <linklist/linklist.h>

DLinkedList dll_new(int (*cmp)(const void *,const void *));
```


Parameters

```
int (*cmp)(const void *,const void *)
```

a comparison function for use in searches of the linked list, may be DLL_NULL_CMP for unordered lists

Description

This routine creates a new list structure. If the list is to have some intrinsic order, a function defining that order should be passed. The comparison routine is called with two node data pointers as the arguments. It must return ‘-1’, ‘0’, or ‘1’ if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns a pointer to a new linked list, or NULL if it can’t allocate it.

3.2.21 dll_next

Retrieve the data in the node following a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

void *dll_next(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

```
DLinkedList ull
```

the list which contains the node

```
DLLNode unode
```

the preceding node

Description

Retrieve the data in the node following a given node in a list.

Returns

It returns NULL if there are no more nodes, else a pointer to the data.

3.2.22 dll_next_node

Get a handle to the node following a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

DLLNode dll_next_node(
```

```
    DLinkedList ull,  
    DLLNode unode  
);
```

Parameters

```
DLinkedList ull  
    the list which contains the node  
  
DLLNode unode  
    the preceding node
```

Description

Get a handle to the node following a given node in a list.

Returns

It returns NULL if there are no more nodes, else a handle to the node.

3.2.23 `dll_node_get_data`

Retrieve the data from a specified linked list node.

Synopsis

```
#include <linklist/linklist.h>  
  
void *dll_node_get_data(DLLNode unode);
```

Parameters

```
DLLNode unode  
    the node in question
```

Description

Retrieve the data from a specified linked list node.

Returns

It returns the node's data pointer.

3.2.24 `dll_node_put_data`

Replace the data pointer in a given node with another.

Synopsis

```
#include <linklist/linklist.h>  
  
void dll_node_put_data(  
    DLLNode unode,  
    void *data  
);
```

Parameters

`DLLNode unode`
the node in question

`void *data`
the new data

Description

This function allows the calling routine to change the data that a node points to. No check is made to insure that any existing ordering of the data is maintained.

3.2.25 `dll_prev`

Retrieve the data in the node preceding a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

void *dll_prev(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

`DLinkedList ull`
the list which contains the node

`DLLNode unode`
the following node

Description

Retrieve the data in the node preceding a given node in a list.

Returns

It returns NULL if there are no more nodes, else a pointer to the data.

3.2.26 `dll_prev_node`

Get a handle to the node preceding a given node in a list.

Synopsis

```
#include <linklist/linklist.h>

DLLNode dll_prev_node(
    DLinkedList ull,
    DLLNode unode
);
```

Parameters

`DLinkedList ull`
the list which contains the node

`DLLNode unode`
the following node

Description

Get a handle to the node preceding a given node in a list.

Returns

It returns `NULL` if there are no more nodes, else a handle to the node.

3.2.27 dll_search

Search a linked list for a node with equivalent data.

Synopsis

```
#include <linklist/linklist.h>

void *dll_search(
    DLinkedList ull,
    void *data,
    int (*cmp)(const void *,const void *)
);
```

Parameters

`DLinkedList ull`
the list to search

`void *data`
the data to search for

`int (*cmp)(const void *,const void *)`
The address of a comparison function. Set to `DLL_NULL_CMP` to use the list's initial comparison function.

Description

This routine searches the list for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the list was initialized. If *that* is not available, a segmentation violation is unavoidable. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return `-1`, `0`, or `1` if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns the node's data pointer if the node was found, `NULL` otherwise.

3.2.28 `dll_search_node`

Search a linked list for a node with equivalent data.

Synopsis

```
#include <linklist/linklist.h>

DLLNode dll_search_node(
    DLinkedList ull,
    void *data,
    int (*cmp)(const void *,const void *)
);
```

Parameters

```
DLinkedList ull
    the list to search

void *data
    the data to search for

int (*cmp)(const void *,const void *)
    The address of a comparison function. Set to DLL_NULL_CMP to use
    the list's initial comparison function.
```

Description

This searches the list for the node for which the passed data compares equivalently with the node's data. It uses the passed comparison routine, if available. If not, it uses that with which the list was initialized. If *that* is not available, a segmentation violation is unavoidable. In the former case the passed data need not have the same form as the data stored in the node.

The comparison routine is called with the passed data as the first argument and the node's data as the second argument. It must return `-1`, `0`, or `1` if, respectively, the first argument is less than, equal to, or greater than the second.

Returns

It returns a handle to the matching node, or `NULL` if not found.

3.2.29 `dll_sizeof_node`

Get the size of a node.

Synopsis

```
#include <linklist/linklist.h>

size_t dll_sizeof_node(void);
```

Description

This routine returns the size of the internal structure used for each node. This does not include user supplied data. Please note that this is a function call, not a macro!.

3.2.30 `dll_tail`

Get the data of the tail node of a list.

Synopsis

```
#include <linklist/linklist.h>

void *dll_tail(DLinkList ull);
```

Parameters

```
DLinkList ull
           the list in question
```

Description

Get the data of the tail node of a list.

Returns

It returns the data of the tail node of a list, NULL if the list is empty.

3.2.31 `dll_tail_node`

Get a node handle to the tail node of a list.

Synopsis

```
#include <linklist/linklist.h>

DLLNode dll_tail_node(DLinkList ull);
```

Parameters

```
DLinkList ull
           the list in question
```

Description

Get a node handle to the tail node of a list.

Returns

It returns the node handle of the tail node of a list, NULL if the list is empty.

3.2.32 `dll_traverse`

Traverse a doubly linked list either forwards or backwards, processing each node.

Synopsis

```
#include <linklist/linklist.h>

int dll_traverse(
    DLinkedList ull,
    int (*action)(void *),
    LinkTraverseDirection direction
);
```

Parameters

DLinkedList ull
A handle to the list to traverse

int (*action)(void *)
The user supplied action routine applied to each node

LinkTraverseDirection direction
the direction in which to traverse the list

Possible values for a **LinkTraverseDirection** are as follows: **HEAD_TO_TAIL**, **TAIL_TO_HEAD**

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer. If an invocation of the action routine returns non-zero, the traversal is aborted and the action routine's return value is returned.

Compare this to [Section 3.2.33 \[dll_traverse_d\]](#), page 43, [Section 3.2.34 \[dll_utrace\]](#), page 44, and [Section 3.2.35 \[dll_utrace_d\]](#), page 45.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.2.33 dll_traverse_d

Traverse a doubly linked list either forwards or backwards, processing each node.

Synopsis

```
#include <linklist/linklist.h>

int dll_traverse_d(
    DLinkedList ull,
    int (*action)(void *),
    LinkTraverseDirection direction,
    void *ndata
);
```

Parameters

`DLinkedList ull`
A handle to the list to traverse

`int (*action)(void *)`
The user supplied action routine applied to each node

`LinkTraverseDirection direction`
the direction in which to traverse the list

Possible values for a `LinkTraverseDirection` are as follows: `HEAD_TO_TAIL`, `TAIL_TO_HEAD`

`void *ndata`
a pointer to a `void *` variable which will receive the data pointer of the node which caused the traversal to be aborted. The type really is `void **`.

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer. If an invocation of the action routine returns non-zero, the traversal is aborted, the current node's data pointer is stored in the location specified by the `ndata` argument, and the action routine's return value is returned.

Compare this to [Section 3.2.32 \[dll_traverse\]](#), page 42, [Section 3.2.34 \[dll_utrace\]](#), page 44, and [Section 3.2.35 \[dll_utrace_d\]](#), page 45.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.2.34 dll_utrace

Traverse a doubly linked list either forwards or backwards, processing each node.

Synopsis

```
#include <linklist/linklist.h>

int dll_utrace(
    DLinkedList ull,
    int (*action)(void *node,void *udata),
    void *udata,
    LinkTraverseDirection direction
);
```

Parameters

`DLinkedList ull`
A handle to the list to traverse


```

int (*action)(void *node,void *udata)
    The user supplied action routine applied to each node

void *udata
    a pointer to data to be passed to the action routine

LinkTraverseDirection direction
    the direction in which to traverse the list

Possible values for a LinkTraverseDirection are as follows: HEAD_
TO_TAIL, TAIL_TO_HEAD

```

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer as well as a pointer provided by the calling routine, thus allowing arbitrary data to be available to the action routine. If an invocation of the action routine returns non-zero the traversal is aborted and the action routine's return value is returned.

Compare this to [Section 3.2.32 \[dll_traverse\]](#), page 42, [Section 3.2.33 \[dll_traverse_d\]](#), page 43, and [Section 3.2.35 \[dll_utraverse_d\]](#), page 45.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.

3.2.35 dll_utraverse_d

Traverse a doubly linked list either forwards or backwards, processing each node.

Synopsis

```

#include <linklist/linklist.h>

int dll_utraverse_d(
    DLinkedList ull,
    int (*action)(void *node,void *udata),
    void *udata,
    LinkTraverseDirection direction,
    void *ndata
);

```

Parameters

```

DLinkedList ull
    A handle to the list to traverse

int (*action)(void *node,void *udata)
    The user supplied action routine applied to each node

void *udata
    a pointer to data to be passed to the action routine

```

LinkTraverseDirection direction

the direction in which to traverse the list

Possible values for a **LinkTraverseDirection** are as follows: **HEAD_TO_TAIL**, **TAIL_TO_HEAD**

void *ndata

a pointer to a **void *** variable which will receive the data pointer of the node which caused the traversal to be aborted. The type really is **void ****.

Description

This routine walks along a list, calling a user supplied action function at each node. The action routine is passed the node's data pointer as well as a pointer provided by the calling routine, thus allowing arbitrary data to be available to the action routine. If an invocation of the action routine returns non-zero, the traversal is aborted, the current node's data pointer is stored in the location specified by the **ndata** argument, and the action routine's return value is returned.

Compare this to [Section 3.2.32 \[dll_traverse\]](#), page 42, [Section 3.2.33 \[dll_traverse_d\]](#), page 43, and [Section 3.2.34 \[dll_utrace\]](#), page 44.

Returns

If the action routines all return '0', it returns '0', else it returns the value returned by the last action routine called.