Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 22.4.5 Manual Pages on command 'LIST_FIRST.3'

$ man LIST_FIRST.3

QUEUE(3) BSD Library Functions Manual QUEUE(3)

NAME

SLIST_EMPTY, SLIST_ENTRY, SLIST_FIRST, SLIST_FOREACH, SLIST_HEAD,

SLIST_HEAD_INITIALIZER, SLIST_INIT, SLIST_INSERT_AFTER, SLIST_INSERT_HEAD,

SLIST_NEXT, SLIST_REMOVE_HEAD, SLIST_REMOVE, STAILQ_CONCAT, STAILQ_EMPTY,

STAILQ_ENTRY, STAILQ_FIRST, STAILQ_FOREACH, STAILQ_HEAD, STAILQ_HEAD_INITIALIZER,

STAILQ_INIT, STAILQ_INSERT_AFTER, STAILQ_INSERT_HEAD, STAILQ_INSERT_TAIL,

STAILQ_NEXT, STAILQ_REMOVE_HEAD, STAILQ_REMOVE, LIST_EMPTY, LIST_ENTRY, LIST_FIRST,

LIST_FOREACH, LIST_HEAD, LIST_HEAD_INITIALIZER, LIST_INIT, LIST_INSERT_AFTER,

LIST_INSERT_BEFORE, LIST_INSERT_HEAD, LIST_NEXT, LIST_REMOVE, TAILQ_CONCAT,

TAILQ_EMPTY, TAILQ_ENTRY, TAILQ_FIRST, TAILQ_FOREACH, TAILQ_FOREACH_REVERSE,

TAILQ_HEAD, TAILQ_HEAD_INITIALIZER, TAILQ_INIT, TAILQ_INSERT_AFTER,

TAILQ_INSERT_BEFORE, TAILQ_INSERT_HEAD, TAILQ_INSERT_TAIL, TAILQ_LAST, TAILQ_NEXT,

TAILQ_PREV, TAILQ_REMOVE, TAILQ_SWAP ? implementations of singly-linked lists,

singly-linked tail queues, lists and tail queues

SYNOPSIS

#include <sys/queue.h>

SLIST_EMPTY(SLIST_HEAD *head);

SLIST_ENTRY(TYPE);

SLIST_FIRST(SLIST_HEAD *head);

SLIST_FOREACH(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);

SLIST_HEAD(HEADNAME, TYPE);

SLIST_HEAD_INITIALIZER(SLIST_HEAD head);

SLIST_INIT(SLIST_HEAD *head);

SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);

SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY NAME);

SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);

SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);

SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);

STAILQ_CONCAT(STAILQ_HEAD *head1, STAILQ_HEAD *head2);

STAILQ_EMPTY(STAILQ_HEAD *head);

STAILQ_ENTRY(TYPE);

STAILQ_FIRST(STAILQ_HEAD *head);

STAILQ_FOREACH(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);

STAILQ_HEAD(HEADNAME, TYPE);

STAILQ_HEAD_INITIALIZER(STAILQ_HEAD head);

STAILQ_INIT(STAILQ_HEAD *head);

STAILQ_INSERT_AFTER(STAILQ_HEAD *head, TYPE *listelm, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_INSERT_HEAD(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_INSERT_TAIL(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_NEXT(TYPE *elm, STAILQ_ENTRY NAME);

STAILQ_REMOVE_HEAD(STAILQ_HEAD *head, STAILQ_ENTRY NAME);

STAILQ_REMOVE(STAILQ_HEAD *head, TYPE *elm, TYPE, STAILQ_ENTRY NAME);

LIST_EMPTY(LIST_HEAD *head);

LIST_ENTRY(TYPE);

LIST_FIRST(LIST_HEAD *head);

LIST_FOREACH(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);

LIST_HEAD(HEADNAME, TYPE);

LIST_HEAD_INITIALIZER(LIST_HEAD head);

LIST_INIT(LIST_HEAD *head);

LIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);

LIST_INSERT_BEFORE(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);

LIST_INSERT_HEAD(LIST_HEAD *head, TYPE *elm, LIST_ENTRY NAME);

LIST_NEXT(TYPE *elm, LIST_ENTRY NAME);

LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME);

LIST_SWAP(LIST_HEAD *head1, LIST_HEAD *head2, TYPE, LIST_ENTRY NAME);

TAILQ_CONCAT(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TAILQ_ENTRY NAME);

TAILQ_EMPTY(TAILQ_HEAD *head);

TAILQ_ENTRY(TYPE);

TAILQ_FIRST(TAILQ_HEAD *head);

TAILQ_FOREACH(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);

TAILQ_FOREACH_REVERSE(TYPE *var, TAILQ_HEAD *head, HEADNAME, TAILQ_ENTRY NAME);

TAILQ_HEAD(HEADNAME, TYPE);

TAILQ_HEAD_INITIALIZER(TAILQ_HEAD head);

TAILQ_INIT(TAILQ_HEAD *head);

TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_INSERT_BEFORE(TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_LAST(TAILQ_HEAD *head, HEADNAME);

TAILQ_NEXT(TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_PREV(TYPE *elm, HEADNAME, TAILQ_ENTRY NAME);

TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);

TAILQ_SWAP(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TYPE, TAILQ_ENTRY NAME);

DESCRIPTION

These macros define and operate on four types of data structures: singly-linked lists, singly-linked tail queues, lists, and tail queues. All four structures sup? port the following functionality:

1. Insertion of a new entry at the head of the list.
2. Insertion of a new entry after any element in the list.
3. O(1) removal of an entry from the head of the list.
4. Forward traversal through the list.
5. Swapping the contents of two lists.

Singly-linked lists are the simplest of the four data structures and support only the above functionality. Singly-linked lists are ideal for applications with large datasets and few or no removals, or for implementing a LIFO queue. Singly-linked lists add the following functionality:

1. O(n) removal of any entry in the list.

Singly-linked tail queues add the following functionality:

1. Entries can be added at the end of a list.
2. O(n) removal of any entry in the list.
3. They may be concatenated.

However:

1. All list insertions must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15% greater and operations run about 20% slower than singly-linked lists.

Singly-linked tail queues are ideal for applications with large datasets and few or no removals, or for implementing a FIFO queue.

All doubly linked types of data structures (lists and tail queues) additionally al? low:

1. Insertion of a new entry before any element in the list.
2. O(1) removal of any entry in the list.

However:

1. Each element requires two pointers rather than one.
2. Code size and execution time of operations (except for removal) is about twice that of the singly-linked data-structures.

Linked lists are the simplest of the doubly linked data structures. They add the following functionality over the above:

1. They may be traversed backwards.

However:

1. To traverse backwards, an entry to begin the traversal and the list in which it is contained must be specified.

Tail queues add the following functionality:

1. Entries can be added at the end of a list.
2. They may be traversed backwards, from tail to head.
3. They may be concatenated.

However:

1. All list insertions and removals must specify the head of the list.
2. Each head entry requires two pointers rather than one.
3. Code size is about 15% greater and operations run about 20% slower than

singly-linked lists.

In the macro definitions, TYPE is the name of a user defined structure, that must contain a field of type SLIST_ENTRY, STAILQ_ENTRY, LIST_ENTRY, or TAILQ_ENTRY, named NAME. The argument HEADNAME is the name of a user defined structure that must be de? clared using the macros SLIST_HEAD, STAILQ_HEAD, LIST_HEAD, or TAILQ_HEAD. See the examples below for further explanation of how these macros are used.

Singly-linked lists

A singly-linked list is headed by a structure defined by the SLIST_HEAD macro. This structure contains a single pointer to the first element on the list. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) removal for arbitrary elements. New elements can be added to the list after an existing element or at the head of the list. An SLIST_HEAD structure is declared as follows:

```
SLIST_HEAD(HEADNAME, TYPE) head;
```

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names head and headp are user selectable.)

The macro SLIST_HEAD_INITIALIZER evaluates to an initializer for the list head.

The macro SLIST_EMPTY evaluates to true if there are no elements in the list.

The macro SLIST_ENTRY declares a structure that connects the elements in the list.

The macro SLIST_FIRST returns the first element in the list or NULL if the list is empty.

The macro SLIST_FOREACH traverses the list referenced by head in the forward direc? tion, assigning each element in turn to var.

The macro SLIST_INIT initializes the list referenced by head.

The macro SLIST_INSERT_HEAD inserts the new element elm at the head of the list.

The macro SLIST_INSERT_AFTER inserts the new element elm after the element listelm.

The macro SLIST_NEXT returns the next element in the list.

The macro SLIST_REMOVE_HEAD removes the element elm from the head of the list. For optimum efficiency, elements being removed from the head of the list should explic? itly use this macro instead of the generic SLIST_REMOVE macro.

The macro SLIST_REMOVE removes the element elm from the list.

Singly-linked list example

```
SLIST_HEAD(slisthead, entry) head =
    SLIST_HEAD_INITIALIZER(head);
struct slisthead *headp;                /* Singly-linked List
                                          head. */
struct entry {
        ...
        SLIST_ENTRY(entry) entries;     /* Singly-linked List. */
        ...
} *n1, *n2, *n3, *np;
SLIST_INIT(&head);                      /* Initialize the list. */
n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
SLIST_INSERT_HEAD(&head, n1, entries);
n2 = malloc(sizeof(struct entry));      /* Insert after. */
SLIST_INSERT_AFTER(n1, n2, entries);
SLIST_REMOVE(&head, n2, entry, entries);/* Deletion. */
free(n2);
n3 = SLIST_FIRST(&head);
SLIST_REMOVE_HEAD(&head, entries);      /* Deletion from the head. */
free(n3);
                                        /* Forward traversal. */
SLIST_FOREACH(np, &head, entries)
        np-> ...
while (!SLIST_EMPTY(&head)) {           /* List Deletion. */
        n1 = SLIST_FIRST(&head);
        SLIST_REMOVE_HEAD(&head, entries);
        free(n1);
}
```

Singly-linked tail queues

A singly-linked tail queue is headed by a structure defined by the STAILQ_HEAD macro. This structure contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are singly linked for minimum space and pointer manipulation overhead at the expense of O(n) re?

moval for arbitrary elements. New elements can be added to the tail queue after an existing element, at the head of the tail queue, or at the end of the tail queue. A STAILQ_HEAD structure is declared as follows:

STAILQ_HEAD(HEADNAME, TYPE) head;

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

struct HEADNAME *headp;

(The names head and headp are user selectable.)

The macro STAILQ_HEAD_INITIALIZER evaluates to an initializer for the tail queue head.

The macro STAILQ_CONCAT concatenates the tail queue headed by head2 onto the end of the one headed by head1 removing all entries from the former.

The macro STAILQ_EMPTY evaluates to true if there are no items on the tail queue.

The macro STAILQ_ENTRY declares a structure that connects the elements in the tail queue.

The macro STAILQ_FIRST returns the first item on the tail queue or NULL if the tail queue is empty.

The macro STAILQ_FOREACH traverses the tail queue referenced by head in the forward direction, assigning each element in turn to var.

The macro STAILQ_INIT initializes the tail queue referenced by head.

The macro STAILQ_INSERT_HEAD inserts the new element elm at the head of the tail queue.

The macro STAILQ_INSERT_TAIL inserts the new element elm at the end of the tail queue.

The macro STAILQ_INSERT_AFTER inserts the new element elm after the element listelm.

The macro STAILQ_NEXT returns the next item on the tail queue, or NULL this item is the last.

The macro STAILQ_REMOVE_HEAD removes the element at the head of the tail queue. For optimum efficiency, elements being removed from the head of the tail queue should use this macro explicitly rather than the generic STAILQ_REMOVE macro.

The macro STAILQ_REMOVE removes the element elm from the tail queue.

Singly-linked tail queue example

```
STAILQ_HEAD(stailhead, entry) head =
    STAILQ_HEAD_INITIALIZER(head);
struct stailhead *headp;                /* Singly-linked tail queue head. */
struct entry {
        ...
        STAILQ_ENTRY(entry) entries;    /* Tail queue. */
        ...
} *n1, *n2, *n3, *np;
STAILQ_INIT(&head);                     /* Initialize the queue. */
n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
STAILQ_INSERT_HEAD(&head, n1, entries);
n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
STAILQ_INSERT_TAIL(&head, n1, entries);
n2 = malloc(sizeof(struct entry));      /* Insert after. */
STAILQ_INSERT_AFTER(&head, n1, n2, entries);
                                        /* Deletion. */
STAILQ_REMOVE(&head, n2, entry, entries);
free(n2);
                                        /* Deletion from the head. */
n3 = STAILQ_FIRST(&head);
STAILQ_REMOVE_HEAD(&head, entries);
free(n3);
                                        /* Forward traversal. */
STAILQ_FOREACH(np, &head, entries)
        np-> ...
                                        /* TailQ Deletion. */
while (!STAILQ_EMPTY(&head)) {
        n1 = STAILQ_FIRST(&head);
        STAILQ_REMOVE_HEAD(&head, entries);
        free(n1);
}
                                        /* Faster TailQ Deletion. */
```

```
n1 = STAILQ_FIRST(&head);
while (n1 != NULL) {
        n2 = STAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
}
STAILQ_INIT(&head);
```

Lists

A list is headed by a structure defined by the LIST_HEAD macro. This structure con? tains a single pointer to the first element on the list. The elements are doubly linked so that an arbitrary element can be removed without traversing the list. New elements can be added to the list after an existing element, before an existing ele? ment, or at the head of the list. A LIST_HEAD structure is declared as follows:

```
LIST_HEAD(HEADNAME, TYPE) head;
```

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the list. A pointer to the head of the list can later be declared as:

```
struct HEADNAME *headp;
```

(The names head and headp are user selectable.)

The macro LIST_HEAD_INITIALIZER evaluates to an initializer for the list head.

The macro LIST_EMPTY evaluates to true if there are no elements in the list.

The macro LIST_ENTRY declares a structure that connects the elements in the list.

The macro LIST_FIRST returns the first element in the list or NULL if the list is empty.

The macro LIST_FOREACH traverses the list referenced by head in the forward direc? tion, assigning each element in turn to var.

The macro LIST_INIT initializes the list referenced by head.

The macro LIST_INSERT_HEAD inserts the new element elm at the head of the list.

The macro LIST_INSERT_AFTER inserts the new element elm after the element listelm.

The macro LIST_INSERT_BEFORE inserts the new element elm before the element listelm.

The macro LIST_NEXT returns the next element in the list, or NULL if this is the last.

The macro LIST_REMOVE removes the element elm from the list.

List example

```
LIST_HEAD(listhead, entry) head =
    LIST_HEAD_INITIALIZER(head);
struct listhead *headp;                 /* List head. */
struct entry {
        ...
        LIST_ENTRY(entry) entries;      /* List. */
        ...
} *n1, *n2, *n3, *np, *np_temp;
LIST_INIT(&head);                       /* Initialize the list. */
n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
LIST_INSERT_HEAD(&head, n1, entries);
n2 = malloc(sizeof(struct entry));      /* Insert after. */
LIST_INSERT_AFTER(n1, n2, entries);
n3 = malloc(sizeof(struct entry));      /* Insert before. */
LIST_INSERT_BEFORE(n2, n3, entries);
LIST_REMOVE(n2, entries);               /* Deletion. */
free(n2);
                                        /* Forward traversal. */
LIST_FOREACH(np, &head, entries)
        np-> ...
while (!LIST_EMPTY(&head)) {            /* List Deletion. */
        n1 = LIST_FIRST(&head);
        LIST_REMOVE(n1, entries);
        free(n1);
}
n1 = LIST_FIRST(&head);                 /* Faster List Deletion. */
while (n1 != NULL) {
        n2 = LIST_NEXT(n1, entries);
        free(n1);
        n1 = n2;
}
LIST_INIT(&head);
```

Tail queues

A tail queue is headed by a structure defined by the TAILQ_HEAD macro. This struc? ture contains a pair of pointers, one to the first element in the tail queue and the other to the last element in the tail queue. The elements are doubly linked so that an arbitrary element can be removed without traversing the tail queue. New elements can be added to the tail queue after an existing element, before an existing element, at the head of the tail queue, or at the end of the tail queue. A TAILQ_HEAD struc? ture is declared as follows:

TAILQ_HEAD(HEADNAME, TYPE) head;

where HEADNAME is the name of the structure to be defined, and TYPE is the type of the elements to be linked into the tail queue. A pointer to the head of the tail queue can later be declared as:

struct HEADNAME *headp;

(The names head and headp are user selectable.)

The macro TAILQ_HEAD_INITIALIZER evaluates to an initializer for the tail queue head.

The macro TAILQ_CONCAT concatenates the tail queue headed by head2 onto the end of the one headed by head1 removing all entries from the former.

The macro TAILQ_EMPTY evaluates to true if there are no items on the tail queue.

The macro TAILQ_ENTRY declares a structure that connects the elements in the tail queue.

The macro TAILQ_FIRST returns the first item on the tail queue or NULL if the tail queue is empty.

The macro TAILQ_FOREACH traverses the tail queue referenced by head in the forward direction, assigning each element in turn to var. var is set to NULL if the loop completes normally, or if there were no elements.

The macro TAILQ_FOREACH_REVERSE traverses the tail queue referenced by head in the reverse direction, assigning each element in turn to var.

The macro TAILQ_INIT initializes the tail queue referenced by head.

The macro TAILQ_INSERT_HEAD inserts the new element elm at the head of the tail queue.

The macro TAILQ_INSERT_TAIL inserts the new element elm at the end of the tail queue.

The macro TAILQ_INSERT_AFTER inserts the new element elm after the element listelm.

The macro TAILQ_INSERT_BEFORE inserts the new element elm before the element listelm.

The macro TAILQ_LAST returns the last item on the tail queue. If the tail queue is empty the return value is NULL.

The macro TAILQ_NEXT returns the next item on the tail queue, or NULL if this item is the last.

The macro TAILQ_PREV returns the previous item on the tail queue, or NULL if this item is the first.

The macro TAILQ_REMOVE removes the element elm from the tail queue.

The macro TAILQ_SWAP swaps the contents of head1 and head2.

Tail queue example

```
TAILQ_HEAD(tailhead, entry) head =
    TAILQ_HEAD_INITIALIZER(head);
struct tailhead *headp;             /* Tail queue head. */
struct entry {
        ...
        TAILQ_ENTRY(entry) entries;     /* Tail queue. */
        ...
} *n1, *n2, *n3, *np;
TAILQ_INIT(&head);                  /* Initialize the queue. */
n1 = malloc(sizeof(struct entry));      /* Insert at the head. */
TAILQ_INSERT_HEAD(&head, n1, entries);
n1 = malloc(sizeof(struct entry));      /* Insert at the tail. */
TAILQ_INSERT_TAIL(&head, n1, entries);
n2 = malloc(sizeof(struct entry));      /* Insert after. */
TAILQ_INSERT_AFTER(&head, n1, n2, entries);
n3 = malloc(sizeof(struct entry));      /* Insert before. */
TAILQ_INSERT_BEFORE(n2, n3, entries);
TAILQ_REMOVE(&head, n2, entries);       /* Deletion. */
free(n2);
                                    /* Forward traversal. */
TAILQ_FOREACH(np, &head, entries)
        np-> ...
                                    /* Reverse traversal. */
TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
```

```
        np-> ...
                                /* TailQ Deletion. */
    while (!TAILQ_EMPTY(&head)) {
        n1 = TAILQ_FIRST(&head);
        TAILQ_REMOVE(&head, n1, entries);
        free(n1);
    }
                                /* Faster TailQ Deletion. */
    n1 = TAILQ_FIRST(&head);
    while (n1 != NULL) {
        n2 = TAILQ_NEXT(n1, entries);
        free(n1);
        n1 = n2;
    }
    TAILQ_INIT(&head);
    n2 = malloc(sizeof(struct entry));  /* Insert before. */
    CIRCLEQ_INSERT_BEFORE(&head, n1, n2, entries);
                              /* Forward traversal. */
    for (np = head.cqh_first; np != (void *)&head;
        np = np->entries.cqe_next)
      np-> ...
                              /* Reverse traversal. */
    for (np = head.cqh_last; np != (void *)&head; np = np->entries.cqe_prev)
      np-> ...
                              /* Delete. */
    while (head.cqh_first != (void *)&head)
      CIRCLEQ_REMOVE(&head, head.cqh_first, entries);
```

CONFORMING TO

Not in POSIX.1, POSIX.1-2001 or POSIX.1-2008. Present on the BSDs. queue functions first appeared in 4.4BSD.

SEE ALSO

insque(3)

COLOPHON

This page is part of release 5.05 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.