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

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'tdelete.3p' command

$ man tdelete.3p

TDELETE(3P) POSIX Programmer's Manual TDELETE(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

tdelete, tfind, tsearch, twalk ? manage a binary search tree

SYNOPSIS

```
#include <search.h>
void *tdelete(const void *restrict key, void **restrict rootp,
    int(*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
    int(*compar)(const void *, const void *));
void *tsearch(const void *key, void **rootp,
    int (*compar)(const void *, const void *));
void twalk(const void *root,
    void (*action)(const void *, VISIT, int));
```

DESCRIPTION

The tdelete(), tfind(), tsearch(), and twalk() functions manipulate bi? nary search trees. Comparisons are made with a user-supplied routine, the address of which is passed as the compar argument. This routine is called with two arguments, which are the pointers to the elements being

compared. The application shall ensure that the user-supplied routine returns an integer less than, equal to, or greater than 0, according to whether the first argument is to be considered less than, equal to, or greater than the second argument. The comparison function need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

The tsearch() function shall build and access the tree. The key argu? ment is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by key, a pointer to this found node shall be returned. Otherwise, the value pointed to by key shall be inserted (that is, a new node is created and the value of key is copied to this node), and a pointer to this node returned. Only pointers are copied, so the application shall ensure that the calling routine stores the data. The rootp argument points to a variable that points to the root node of the tree. A null pointer value for the variable pointed to by rootp denotes an empty tree; in this case, the variable shall be set to point to the node which shall be at the root of the new tree.

Like tsearch(), tfind() shall search for a node in the tree, returning a pointer to it if found. However, if it is not found, tfind() shall return a null pointer. The arguments for tfind() are the same as for tsearch().

The tdelete() function shall delete a node from a binary search tree. The arguments are the same as for tsearch(). The variable pointed to by rootp shall be changed if the deleted node was the root of the tree. If the deleted node was the root of the tree and had no children, the variable pointed to by rootp shall be set to a null pointer. The tdelete() function shall return a pointer to the parent of the deleted node, or an unspecified non-null pointer if the deleted node was the root node, or a null pointer if the node is not found.

If tsearch() adds an element to a tree, or tdelete() successfully deletes an element from a tree, the concurrent use of that tree in an? other thread, or use of pointers produced by a previous call to tfind()

or tsearch(), produces undefined results.

The twalk() function shall traverse a binary search tree. The root argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument action is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument shall be the address of the node being visited. The structure pointed to by this argument is unspecified and shall not be modified by the application, but it shall be possible to cast a pointer-to-node into a pointer-to-pointer-to-element to access the element stored in the node. The second argument shall be a value from an enumeration data type:

```
typedef enum { preorder, postorder, endorder, leaf } VISIT;
```

(defined in <search.h>), depending on whether this is the first, second, or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument shall be the level of the node in the tree, with the root being level 0.

If the calling function alters the pointer to the root, the result is undefined.

If the functions pointed to by action or compar (for any of these binary search functions) change the tree, the results are undefined.

These functions are thread-safe only as long as multiple threads do not access the same tree.

RETURN VALUE

If the node is found, both tsearch() and tfind() shall return a pointer to it. If not, tfind() shall return a null pointer, and tsearch() shall return a pointer to the inserted item.

A null pointer shall be returned by tsearch() if there is not enough space available to create a new node.

A null pointer shall be returned by tdelete(), tfind(), and tsearch() if rootp is a null pointer on entry.

The tdelete() function shall return a pointer to the parent of the

deleted node, or an unspecified non-null pointer if the deleted node was the root node, or a null pointer if the node is not found.

The twalk() function shall not return a value.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

The following code reads in strings and stores structures containing a pointer to each string and a count of its length. It then walks the tree, printing out the stored strings and their lengths in alphabetical order.

```
#include <limits.h>
#include <search.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
struct element {      /* Pointers to these are stored in the tree. */
    int     count;
    char    string[];
};
void  *root = NULL;          /* This points to the root. */
int main(void)
{
    char   str[_POSIX2_LINE_MAX+1];
    int    length = 0;
    struct element *elementptr;
    void   *node;
    void   print_node(const void *, VISIT, int);
    int    node_compare(const void *, const void *),
           delete_root(const void *, const void *);
    while (fgets(str, sizeof(str), stdin))  {
        /* Set element. */
        length = strlen(str);
```

```
        if (str[length-1] == '\n')
            str[--length] = '\0';
        elementptr = malloc(sizeof(struct element) + length + 1);
        strcpy(elementptr->string, str);
        elementptr->count = 1;
        /* Put element into the tree. */
        node = tsearch((void *)elementptr, &root, node_compare);
        if (node == NULL) {
            fprintf(stderr,
                    "tsearch: Not enough space available\n");
            exit(EXIT_FAILURE);
        }
        else if (*(struct element **)node != elementptr) {
            /* A node containing the element already exists */
            (*(struct element **)node)->count++;
            free(elementptr);
        }
    }
    twalk(root, print_node);
    /* Delete all nodes in the tree */
    while (root != NULL) {
        elementptr = *(struct element **)root;
        printf("deleting node: string = %s,  count = %d\n",
                elementptr->string,
                elementptr->count);
        tdelete((void *)elementptr, &root, delete_root);
        free(elementptr);
    }
    return 0;
}
/*
 *  This routine compares two nodes, based on an
 *  alphabetical ordering of the string field.
```

```
 */
int
node_compare(const void *node1, const void *node2)
{
    return strcmp(((const struct element *) node1)->string,
        ((const struct element *) node2)->string);
}
/*
 *  This comparison routine can be used with tdelete()
 *  when explicitly deleting a root node, as no comparison
 *  is necessary.
 */
int
delete_root(const void *node1, const void *node2)
{
    return 0;
}
/*
 *  This routine prints out a node, the second time
 *  twalk encounters it or if it is a leaf.
 */
void
print_node(const void *ptr, VISIT order, int level)
{
    const struct element *p = *(const struct element **) ptr;
    if (order == postorder || order == leaf)  {
        (void) printf("string = %s,  count = %d\n",
            p->string, p->count);
    }
}
```

APPLICATION USAGE

The root argument to twalk() is one level of indirection less than the rootp arguments to tdelete() and tsearch().

There are two nomenclatures used to refer to the order in which tree nodes are visited. The twalk() function uses preorder, postorder, and endorder to refer respectively to visiting a node before any of its children, after its left child and before its right, and after both its children. The alternative nomenclature uses preorder, inorder, and pos?torder to refer to the same visits, which could result in some confu?sion over the meaning of postorder.

Since the return value of tdelete() is an unspecified non-null pointer in the case that the root of the tree has been deleted, applications should only use the return value of tdelete() as indication of success or failure and should not assume it can be dereferenced. Some implemen?tations in this case will return a pointer to the new root of the tree (or to an empty tree if the deleted root node was the only node in the tree); other implementations return arbitrary non-null pointers.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

hcreate(), lsearch()

The Base Definitions volume of POSIX.1?2017, <search.h>

COPYRIGHT

files to man page format. To report such errors, see https://www.ker?

nel.org/doc/man-pages/reporting_bugs.html .