NAME

tsearch, tfind, tdelete, twalk, twalk_r, tdestroy − manage a binary search tree

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

```
#include <search.h>

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

void *tsearch(const void *key, void **rootp,
        int (*compar)(const void *, const void *));
void *tfind(const void *key, void *const *rootp,
        int (*compar)(const void *, const void *));
void *tdelete(const void *restrict key, void **restrict rootp,
        int (*compar)(const void *, const void *));
void twalk(const void *root,
        void (*action)(const void *nodep, VISIT which,
                int depth));

#define _GNU_SOURCE         /* See feature_test_macros(7) */
#include <search.h>

void twalk_r(const void *root,
        void (*action)(const void *nodep, VISIT which,
                void *closure),
        void *closure);
void tdestroy(void *root, void (*free_node)(void *nodep));
```

DESCRIPTION

tsearch(), **tfind**(), **twalk**(), and **tdelete**() manage a binary search tree. They are generalized from Knuth (6.2.2) Algorithm T. The first field in each node of the tree is a pointer to the corresponding data item. (The calling program must store the actual data.) *compar* points to a comparison routine, which takes pointers to two items. It should return an integer which is negative, zero, or positive, depending on whether the first item is less than, equal to, or greater than the second.

tsearch() searches the tree for an item. *key* points to the item to be searched for. *rootp* points to a variable which points to the root of the tree. If the tree is empty, then the variable that *rootp* points to should be set to NULL. If the item is found in the tree, then **tsearch**() returns a pointer to the corresponding tree node. (In other words, **tsearch**() returns a pointer to a pointer to the data item.) If the item is not found, then **tsearch**() adds it, and returns a pointer to the corresponding tree node.

tfind() is like **tsearch**(), except that if the item is not found, then **tfind**() returns NULL.

tdelete() deletes an item from the tree. Its arguments are the same as for **tsearch**().

twalk() performs depth-first, left-to-right traversal of a binary tree. *root* points to the starting node for the traversal. If that node is not the root, then only part of the tree will be visited. **twalk**() calls the user function *action* each time a node is visited (that is, three times for an internal node, and once for a leaf). *action*, in turn, takes three arguments. The first argument is a pointer to the node being visited. The structure of the node is unspecified, but it is possible to cast the pointer to a pointer-to-pointer-to-element in order to access the element stored within the node. The application must not modify the structure pointed to by this argument. The second argument is an integer which takes one of the values **preorder**, **postorder**, or **endorder** depending on whether this is the first, second, or third visit to the internal node, or the value **leaf** if this is the single visit to a leaf node. (These symbols are defined in *<search.h>*.) The third argument is the depth of the node; the root node has depth zero.

(More commonly, **preorder**, **postorder**, and **endorder** are known as **preorder**, **inorder**, and **postorder**: before visiting the children, after the first and before the second, and after visiting the children. Thus, the choice of name **postorder** is rather confusing.)

twalk_r() is similar to **twalk**(), but instead of the *depth* argument, the *closure* argument pointer is passed to each invocation of the action callback, unchanged. This pointer can be used to pass information to and from the callback function in a thread-safe fashion, without resorting to global variables.

tdestroy() removes the whole tree pointed to by *root*, freeing all resources allocated by the **tsearch**() function. For the data in each tree node the function *free_node* is called. The pointer to the data is passed as the argument to the function. If no such work is necessary, *free_node* must point to a function doing nothing.

RETURN VALUE

tsearch() returns a pointer to a matching node in the tree, or to the newly added node, or NULL if there was insufficient memory to add the item. **tfind**() returns a pointer to the node, or NULL if no match is found. If there are multiple items that match the key, the item whose node is returned is unspecified.

tdelete() returns a pointer to the parent of the node deleted, or NULL if the item was not found. If the deleted node was the root node, **tdelete**() returns a dangling pointer that must not be accessed.

tsearch(), **tfind**(), and **tdelete**() also return NULL if *rootp* was NULL on entry.

VERSIONS

twalk_r() is available since glibc 2.30.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
tsearch(), **tfind**(), **tdelete**()	Thread safety	MT-Safe race:rootp
twalk()	Thread safety	MT-Safe race:root
twalk_r()	Thread safety	MT-Safe race:root
tdestroy()	Thread safety	MT-Safe

STANDARDS

POSIX.1-2001, POSIX.1-2008, SVr4. The functions **tdestroy**() and **twalk_r**() are GNU extensions.

NOTES

twalk() takes a pointer to the root, while the other functions take a pointer to a variable which points to the root.

tdelete() frees the memory required for the node in the tree. The user is responsible for freeing the memory for the corresponding data.

The example program depends on the fact that **twalk**() makes no further reference to a node after calling the user function with argument "endorder" or "leaf". This works with the GNU library implementation, but is not in the System V documentation.

EXAMPLES

The following program inserts twelve random numbers into a binary tree, where duplicate numbers are collapsed, then prints the numbers in order.

```
#define _GNU_SOURCE     /* Expose declaration of tdestroy() */
#include <search.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

static void *root = NULL;

static void *
xmalloc(size_t n)
```

```
{
    void *p;

    p = malloc(n);
    if (p)
        return p;
    fprintf(stderr, "insufficient memory\n");
    exit(EXIT_FAILURE);
}

static int
compare(const void *pa, const void *pb)
{
    if (*(int *) pa < *(int *) pb)
        return -1;
    if (*(int *) pa > *(int *) pb)
        return 1;
    return 0;
}

static void
action(const void *nodep, VISIT which, int depth)
{
    int *datap;

    switch (which) {
    case preorder:
        break;
    case postorder:
        datap = *(int **) nodep;
        printf("%6d\n", *datap);
        break;
    case endorder:
        break;
    case leaf:
        datap = *(int **) nodep;
        printf("%6d\n", *datap);
        break;
    }
}

int
main(void)
{
    int  *ptr;
    int  **val;

    srand(time(NULL));
    for (unsigned int i = 0; i < 12; i++) {
        ptr = xmalloc(sizeof(*ptr));
        *ptr = rand() & 0xff;
        val = tsearch(ptr, &root, compare);
        if (val == NULL)
            exit(EXIT_FAILURE);
```

```
            if (*val != ptr)
                free(ptr);
        }
        twalk(root, action);
        tdestroy(root, free);
        exit(EXIT_SUCCESS);
    }
```

SEE ALSO

bsearch(3), **hsearch**(3), **lsearch**(3), **qsort**(3)