NAME

tsearch, tfind, tdelete, twalk – manage binary search trees

SYNOPSIS

#include <search.h>

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** **key*, **void** ***rootp*,
    int (**compar*)(**const void** *, **const void** *));

void twalk(void **root*, **void(****action*) **(void** *, **VISIT, int));**

MT-LEVEL

Safe

DESCRIPTION

tsearch( ), **tfind( )**, **tdelete( )**, and **twalk( )** are routines for manipulating binary search trees. They are generalized from *Knuth (6.2.2) Algorithms T and D.* All comparisons are done with a user-supplied routine. This routine is called with two arguments, the pointers to the elements being compared. It 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.

tsearch( ) is used to build and access the tree. *key* is a pointer to a datum to be accessed or stored. If there is a datum in the tree equal to **key* (the value pointed to by *key*), a pointer to this found datum is returned. Otherwise, **key* is inserted, and a pointer to it returned. Only pointers are copied, so the calling routine must store the data. *rootp* points to a variable that points to the root of the tree. A **NULL** value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the datum which will be at the root of the new tree.

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

tdelete( ) deletes a node from a binary search tree. The arguments are the same as for **tsearch( )**. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. **tdelete( )** returns a pointer to the parent of the deleted node, or a **NULL** pointer if the node is not found.

twalk( ) traverses a binary search tree. *root* is the root of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) *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 is the address of the node being visited. The second argument is a value from an enumeration data type *typedef enum { preorder, postorder, endorder, leaf } VISIT;* (defined in the **<search.h>** header), depending on whether this is the first, second or third time that the node has been visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element, and cast to type pointer-to-character. Similarly, although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

RETURN VALUES

A **NULL** pointer is returned by **tsearch( )** if there is not enough space available to create a new node. A **NULL** pointer is returned by **tfind( )** and **tdelete( )** if *rootp* is **NULL** on entry.

If the datum is found, both **tsearch( )** and **tfind( )** return a pointer to it. If not, **tfind( )** returns **NULL**, and **tsearch( )** returns a pointer to the inserted item.

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 <string.h>
#include <stdio.h>
#include <search.h>

struct node {
        char *string;
        int length;
};
char string_space[10000];
struct node nodes[500];
void *root = NULL;

int node_compare(const void *node1, const void *node2) {
        return strcmp(((const struct node *) node1)->string,
                ((const struct node *) node2)->string);
}

void print_node(void **node, VISIT order, int level) {
        if (order == preorder || order == leaf) {
                printf("length=%d, string=%20s\n",
                (*(struct node **)node)->length,
                (*(struct node **)node)->string);
        }
}

main()
{
        char *strptr = string_space;
        struct node *nodeptr = nodes;
        int i = 0;
```

```
        while (gets(strptr) != NULL && i++ < 500) {
                nodeptr->string = strptr;
                nodeptr->length = strlen(strptr);
                (void) tsearch((void *)nodeptr,
                                &root, node_compare);
                strptr += nodeptr->length + 1;
                nodeptr++;
        }
        twalk(root, print_node);
}
```

SEE ALSO

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

NOTES

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

There are two nomenclatures used to refer to the order in which tree nodes are visited. **tsearch** 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 alternate nomenclature uses preorder, inorder and postorder to refer to the same visits, which could result in some confusion over the meaning of postorder.

If the calling function alters the pointer to the root, results are unpredictable.