NAME

avl_add, **avl_del**, **avl_del_node**, **bst_srch**, **bst_add**, **bst_del**, **bst_del_node** — AVL binary search tree functions

SYNOPSIS

```
#include <avlbst.h>

int
avl_add(struct bst *tree, union bst_val key, union bst_val data);

int
avl_del(struct bst *tree, union bst_val key);

void
avl_del_node(struct bst *tree, struct bst_node *node);

int
bst_srch(struct bst *tree, union bst_val key, struct bst_node **node);

int
bst_add(struct bst *tree, union bst_val key, union bst_val data);

int
bst_del(struct bst *tree, union bst_val key);

void
bst_del_node(struct bst *tree, struct bst_node *node);
```

DESCRIPTION

libavlbst is a general purpose AVL tree implementaion. Each node can store a *key* and a *data* value:

```
struct bst_node {
        union bst_val   key;
        union bst_val   data;
        int             bf;
        struct bst_node *parent, *left, *right;
};
```

For *key* and *data* several types can be choosen:

```
union bst_val {
        void     *p;
        int      i;
        long     l;
        uint64_t u64;
        time_t   t;
};
```

Hence for many uses it may not be necessary to allocate memory for *key* or *data*. All functions operating on the tree use a pointer *tree* to a structure which contains a pointer to the top node and the compare function:

```
struct bst {
        struct bst_node *root;
        int (*cmp)(union bst_val, union bst_val);
};
```

avl_add() adds nodes to the tree. **bst_add**() does the same without rebalancing the tree.

avl_del() deletes nodes from the tree by specifying *key* to identify the node. **bst_del**() does the same without rebalancing the tree. This is useful if nodes should be deletes while walking through the tree. It is *not allowed* to use a **avl_...**() function after using a **bst_...**() function on a tree.

avl_del_node() deletes nodes from the tree by specifying a pointer to the *node*. **bst_del_node**() does the same without rebalancing the tree.

bst_srch() is used to search for an existing *key*. If argument *node* is non-zero a pointer to the node with this key is stored there.

Programs which link against this library should use a linker command similar to

```
LIBDIR = /usr/local/lib
...
$(CC) $(LDFLAGS) -L$(LIBDIR) -Wl,-rpath,$(LIBDIR) -o $@ $(OBJS) \
    -lavlbst
```

RETURN VALUES

avl_add() and **bst_add**() return 0 on success; otherwise the value BST_EEXIST if the key already exists.

avl_del() and **bst_del**() return 0 on success; otherwise the value BST_ENOENT if the key oes not exists.

bst_srch() returns 0 if the node had been found; otherwise a value different from 0.

EXAMPLES

This simple example outputs the sorted command line arguments and then removes each node from the tree:

```
#include <stdio.h>
#include <string.h>
#include <avlbst.h>

static int cmp(union bst_val, union bst_val);

int
main(int argc, char **argv) {
    struct bst args = { NULL, cmp };
    while (argc--)
        avl_add(&args, (union bst_val)(void *)(*argv++),
          (union bst_val)(int)0); /* key also used as data */
    print_sorted_args(args.root);
    while (args.root)
        avl_del_node(&args, args.root);
}

static void
print_sorted_args(struct bst_node *node) {
    if (!node)
        return;
    print_sorted_args(node->left);
    printf("%s\n", (char *)node->key.p);
```

```
    print_sorted_args(node->right);
}

static int
cmp(union bst_val a, union bst_val b) {
    return strcmp(a.p, b.p);
}
```

Non-recursive tree traversal

The following function can be used to perform a non-recursive tree traversal (outputs the same sequence as recursive function **print_sorted_args**() in the example above):

```
void
proctree(struct bst *tree, void (*proc)(struct bst_node *),
  void (*del)(struct bst *, struct bst_node *)) {
    struct bst_node *node, *node2;
    int go_proc;
    if (!(node = tree->root))
        return;
enter_node:
    while (node->left)
        node = node->left;
proc_data:
    proc(node);
    if (node->right) {
        node = node->right;
        goto enter_node;
    }
go_up:
    node2 = node;
    node = node->parent;
    if (node)
        go_proc = node2 == node->left;
    if (del)
        del(tree, node2); /* Must be non-balancing delete! */
    if (!node)
        return;
    if (go_proc)
        goto proc_data;
    goto go_up;
}
```

proc is called for each node found in order of the keys. If *del* is not NULL it is called for each visited node for removing it. This function must use a non-balancing delete.

SEE ALSO

avl_add(3), avl_del(3), avl_del_node(3), bst_srch(3), bst_add(3), bst_del(3), bst_del_node(3)