/*------------------------------------------*/
#include <stdio.h>
#include "tree1.h")

BALANCED (AVL) BINARY TREE ROUTINES

insert - the function insert returns a pointer to the data area of a node in a balanced binary tree which matches the value pointed to by the argument udata. It returns a pointer to the udata structure within the tree which corresponds to the value of udata passed to the routine.

Arguments:
pt  char ** address of a pointer to the head of the binary tree. Note that the routine may change pt's value.
udata char * address of the data area which is to be inserted (or searched for) in the binary tree. The contents of this area are arbitrary, since the caller must also provide a function for determining whether or not data areas are the same. Insert will allocate memory of this size when a new node is formed, and copy the usize bytes of udata to this memory, storing a pointer in the node.
usize int size, in bytes, of the udata structure. Insert will allocate memory of this size when a new node is formed, and copy the usize bytes of udata to this memory, storing a pointer in the node.
comp int (*)() function defining how to compare two user data areas. The function comp will be passed pointers to the udata area already found in the tree (first argument) and the udata area which is being inserted (second argument), and should return -1 if the first argument is less than the second, 0 if they are equal, and 1 if the first argument is greater than the second.

traverse - the function traverse travels to each node of a balanced binary tree, and calls a user-supplied function to operate on the data area.

Arguments:
t  char * pointer to the head of the tree to be traversed.
func int (*)() user-supplied function to be executed on each data area of the tree. The function will be passed a pointer to the data area.

delete - the function delete traverses each node of a balanced binary tree, calls an optional user-supplied routine to delete any memory allocated by the calling program, and then deletes the memory allocated within the node itself, leaving no trace of the original binary tree in memory.

Arguments:
pt  char ** address of the pointer to the head of the balanced binary tree. Delete will set this pointer equal to NULL when the tree is deleted.
delfn int (*)() optional program to delete user-allocated memory. If no deletion of this memory is required, the calling routine should provide the value NULL, defined in stdio.h.

/*----------------------------------------------------------------------
NODE structure used internally for binary tree routines. The calling routine DOES NOT need to ever refer to the NODE structure, although it may want to include this file since it contains prototypes for the binary tree functions.
------------------------------------------------------------------------*/

struct NODE {
  struct NODE *left,*right;
  short bal,dum;
  char *udata;
};

/* Function prototypes - functions defined in btree1.c */
char *insert(char **pt,char *udata,int usize,int (*comp)());
void traverse(char *t,int (*func)());
void delete(char **pt,int (*delfn)());

char *insert(char **pt,char *udata,int usize,int (*comp)())
{
  struct NODE *do_insert();
  static short zero = 0;
  return((char*)((do_insert((struct NODE **)pt,udata,usize,&zero,comp))->udata));
}

traverse(char *t,int (*func)())
{
  struct NODE *s = (struct NODE*)t;
  if(s->left != NULL)traverse((char*)s->left,func);
  (*func)((char*)(s->udata));
  if(s->right != NULL)traverse((char*)s->right,func);
}

delete(char **pt,int (*delfn)())
{
  struct NODE **pn = (struct NODE **)pt;
  if(*pt){
    delete((char**)(&(*pn)->left),delfn);
    delete((char**)(&(*pn)->right),delfn);
    if(delfn)(*delfn)(*pt);
    free((char*)(*pn)->udata);
    free(*pt);
    *pt = NULL;
  }
}
static struct NODE *ret;
struct NODE *do_insert(struct NODE **pt, char *udata, int usize, short *bal,
        int (*comp)())
{
    struct NODE *t1, *t2;
    char *calloc(), *malloc();
    int cc, a, i;

    if (*pt == NULL){
        if ((ret = *pt = (struct NODE*)calloc(1, sizeof(struct NODE))) == NULL){
            fprintf(stderr,"insert: No memory available. Exiting ...
"); exit(1);
        }

        if (((*pt)->udata = malloc((unsigned)usize)) == NULL){
            fprintf(stderr,"insert: No memory available. Exiting ...
"); exit(1);
        }

        for (i = 0; i < usize; i++) (*pt)->udata[i] = udata[i];
        *bal = 1;
        return(ret);
    }

    cc = (*comp)((*pt)->udata, udata);
    if (cc > 0){
        (void) do_insert(&((*pt)->right), udata, usize, bal, comp);
        if (*bal){
            switch((*pt)->bal){
            case -1:
                (*pt)->bal = 0;
                *bal = 0;
                break;
            case 0:
                (*pt)->bal = 1;
                break;
            case 1:
                t1 = (*pt)->right;
                if (t1->bal == 1){
                    (*pt)->right = t1->left;
                    t1->left = *pt;
                    (*pt)->bal = 0;
                    *pt = t1;
                }
                else{
                    t2 = t1->left;
                    t1->left = t2->right;
                    t2->right = t1;
                    (*pt)->right = t2->left;
                    t2->left = *pt;
                    (*pt)->bal = t2->bal == 1 ? -1 : 0;
                    t1->bal = t2->bal == -1 ? 1 : 0;
                    *pt = t2;
                }
                (*pt)->bal = 0;
                *bal = 0;
            }
            (*pt)->bal = 0;
            return(ret);
        }
        return(ret);
    }
    else if (cc < 0){
        (void) do_insert(&((*pt)->left), udata, usize, bal, comp);
    }
    return(ret);
}

if (*bal){
    switch((*pt)->bal){
    case 1:
        (*pt)->bal = 0;
        *bal = 0;
        break;
    case 0:
        (*pt)->bal = -1;
        break;
    case -1:
        t1 = (*pt)->left;
        if (t1->bal == -1){
            (*pt)->left = t1->right;
            t1->right = *pt;
            (*pt)->bal = 0;
            *pt = t1;
        }
        else{
            t2 = t1->right;
            t1->right = t2->left;
            t2->left = t1;
            (*pt)->left = t2->right;
            t2->right = (*pt);
            (*pt)->bal = t2->bal == -1 ? 1 : 0;
            t1->bal = t2->bal == 1 ? -1 : 0;
            *pt = t2;
        }
        (*pt)->bal = 0;
        *bal = 0;
    }
}

/*----------------------------------------*
Example of a comparison function:
----------------------------------------*/

int scmp(char *t1, char *t2){
    struct UDATA *u1 = (struct UDATA*)t1;
    struct UDATA *u2 = (struct UDATA*)t2;
    int i;
    i = strcmp(u1->name, u2->name);
    return(i == 0 ? 0 : i > 0 ? 1 : -1);
}