```
/* CS261- Assignment 4 - bst.c */
/* Name: Jacob Karcz
 * Date: 10.24.2016
 * Solution description: binary search tree function implementation file
 */

/
    ****************************************************************************
    **************
                                    File: bst.h
 ************* Interface definition of the binary search tree data structure
     *************


     ***************************************************************************
     **************/

#ifndef __BST_H
#define __BST_H

/* Defines the type to be stored in the data structure.  These macros
 * are for convenience to avoid having to search and replace/dup code
 * when you want to build a structure of doubles as opposed to ints
 * for example.
 */

# ifndef TYPE
# define TYPE      void*   // TYPE VOID DECLARATION
# endif

/* function used to compare two TYPE values to each other, define this in your
    compare.c file */
int compare(TYPE left, TYPE right);
/* function used to print TYPE values, define this in your compare.c file */
void print_type(TYPE curval);


struct BSTree;
/* Declared in the c source file to hide the structure members from the user.
    */

/* Initialize binary search tree structure. */
void initBSTree(struct BSTree *tree);

/* Alocate and initialize search tree structure. */
struct BSTree *newBSTree();

/* Deallocate nodes in BST. */
void clearBSTree(struct BSTree *tree);

/* Deallocate nodes in BST and deallocate the BST structure. */
void deleteBSTree(struct BSTree *tree);

/
    ****************************************************************************
    **************
 **********************************-- BST Bag interface --
     **********************************

     ****************************************************************************
     ***************/
int  isEmptyBSTree(struct BSTree *tree);
int     sizeBSTree(struct BSTree *tree);

void     addBSTree(struct BSTree *tree, TYPE val);
int containsBSTree(struct BSTree *tree, TYPE val);
void  removeBSTree(struct BSTree *tree, TYPE val);
void printTree(struct BSTree *tree);

#endif /* bst_h */


/
    ****************************************************************************
    **************

 ******************                    File: structs.h
     ****************


     ****************************************************************************
     ***************/
/* You can modify the structure to store whatever you'd like in your BST */

struct data {
    int number;
    char *name;
};

/
    ****************************************************************************
    **************

 ******************                    File: compare.c
     ****************


     ****************************************************************************
     ***************/

#include <stdio.h>
#include <assert.h>
#include "bst.h"
#include "structs.h"

/*-------------------------------------------------------------------------------
 very similar to the compareTo method in java or the strcmp function in c. it
 returns an integer to tell you if the left value is greater then, less then,
     or
 equal to the right value. you are comparing the number variable, letter is not
 used in the comparison.
```

```
     if left < right return -1
     if left > right return 1
     if left = right return 0
     */

    /*Define this function, type casting the value of void * to the desired type.
     The current definition of TYPE in bst.h is void*, which means that left and
     right are void pointers. To compare left and right, you should first cast
     left and right to the corresponding pointer type (struct data *), and then
     compare the values pointed by the casted pointers.

     DO NOT compare the addresses pointed by left and right, i.e. "if (left <
         right)",
     which is really wrong.
     */
    /
        ****************************************************************************
        **************
     *Compare function for data stgruct holding a name and number
     *function: compare(TYPE left, TYPE right)
     *precondition: left and right are not null of TYPE void *
     *parameter(s): left and right bst pointers to void
     *postcondition:  if left < right return -1
     if left > right return 1
     if left = right return 0

         ***************************************************************************
         **************/
    int compare(TYPE left, TYPE right) {
        /*FIXME: write this*/
        assert (left != NULL);
        assert (right != NULL);

        //typeCast TYPE void * as TYPE data
        struct data *leftData;
        struct data *rightData;
        leftData = (struct data *) left;
        rightData = (struct data *) right;

        //compare the values of the data structs
        if (leftData->number < rightData->number)
            return -1;
        else if (leftData->number > rightData->number)
            return 1;
        else
            return 0;
        return 0;

    }

    /*Define this function, type casting the value of void * to the desired type*/
    void print_type(TYPE curval) {
        /*FIXME: write this*/

        struct data *dataPtr;
        dataPtr = (struct data *) curval;
        printf("%s: %d\n", dataPtr->name, dataPtr->number);
    }



    /
        ****************************************************************************
        **************

                                            File: bst.c
     ******************Implementation of the binary search tree data
         structure****************


         ****************************************************************************
         **************/
    #include <stdlib.h>
    #include <stdio.h>
    #include <assert.h>
    #include "bst.h"
    #include "structs.h"


    struct Node {
        TYPE         val;
        struct Node *left;
        struct Node *right;
    };

    struct BSTree {
        struct Node *root;
        int          cnt;
    };

    /
        ****************************************************************************
        **************
     function to initialize the binary search tree.
     param: binary tree
     pre: tree is not null
     post:  tree size is 0
     root is null

         ****************************************************************************
         **************/

    void initBSTree(struct BSTree *tree) {
        tree->cnt  = 0;
        tree->root = 0;
    }

    /
        ****************************************************************************
        **************
     function to create a binary search tree.
     param: none
     pre: none
     post: tree->count = 0
     tree->root = 0;

         ***************************************************************************
         **************/

    struct BSTree*  newBSTree() {
        struct BSTree *tree = (struct BSTree *)malloc(sizeof(struct BSTree));
        assert(tree != 0);

        initBSTree(tree);
        return tree;
    }

    /
        ***************************************************************************
        **************
     function to free the nodes of a binary search tree
     param: node  the root node of the tree to be freed
     pre: none
     post: node and all descendants are deallocated

         ***************************************************************************
         **************/

    void _freeBST(struct Node *node) {
        if (node != 0) {
            _freeBST(node->left);
            _freeBST(node->right);
            free(node);
        }
    }

    /
        ***************************************************************************
        **************
     function to clear the nodes of a binary search tree
     param: tree    a binary search tree
     pre: tree ! = null
     post: the nodes of the tree are deallocated
     tree->root = 0;
     tree->cnt = 0

         ***************************************************************************
         **************/
    void clearBSTree(struct BSTree *tree) {
        _freeBST(tree->root);
        tree->root = 0;
        tree->cnt  = 0;
    }

    /
        ***************************************************************************
        **************
     function to deallocate a dynamically allocated binary search tree
     param: tree   the binary search tree
     pre: tree != null;
     post: all nodes and the tree structure itself are deallocated.

         ***************************************************************************
         **************/
    void deleteBSTree(struct BSTree *tree) {
        clearBSTree(tree);
        free(tree);
    }

    /
        ***************************************************************************
        **************
     function to determine if  a binary search tree is empty.

     param: tree    the binary search tree
     pre:  tree is not null

         ***************************************************************************
         **************/
    int isEmptyBSTree(struct BSTree *tree) {
        assert(tree != NULL);
        return (tree->cnt == 0);
    }

    /
        ***************************************************************************
        **************
     function to determine the size of a binary search tree

     param: tree    the binary search tree
     pre:  tree is not null

         ***************************************************************************
         **************/
    int sizeBSTree(struct BSTree *tree) {
        assert(tree != NULL);
        return tree->cnt;
    }

    /
        ***************************************************************************
        **************
     recursive helper function to add a node to the binary search tree.
     HINT: You have to use the compare() function to compare values.
     param:  cur    the current root node
     val    the value to be added to the binary search tree
     pre:   val is not null

         ***************************************************************************
         **************/
    struct Node *_addNode(struct Node *cur, TYPE val) {
        /*write this*/
        //assert(cur != NULL);
        struct Node * newNode;

        //baseCase
        if (cur == NULL) {                             //found a spot
            newNode = malloc(sizeof(struct Node));     //malloc node
            assert(newNode != 0);                          //check malloc
            newNode->val = val;                        //set newNode value
            newNode->left = 0;
            newNode->right = 0;                        // set leaf's children to
                null
            return newNode;                            //return the new node to
                caller
        }
        //recursive case; current node is not null, compare the value passed to
            current value
        if (compare(val, cur->val) == -1)              //if(val < cur->val)
            cur->left = _addNode (cur->left , val);    //newValue is less than
        else// if (compare(val, cur->val == 1)
            cur->right = _addNode (cur->right, val);   //newValue is >= current
        return cur;          //return current (moded) node to caller to rebuild
            the tree
    }

    /
        ***************************************************************************
        **************
     function to add a value to the binary search tree
     param: tree   the binary search tree
     val        the value to be added to the tree

     pre:   tree is not null
     val is not null
     pose:  tree size increased by 1
     tree now contains the value, val

         ***************************************************************************
         **************/
    void addBSTree(struct BSTree *tree, TYPE val) {
        tree->root = _addNode(tree->root, val);
        tree->cnt++;
    }


    /
        ***************************************************************************
        **************
     function to determine if the binary search tree contains a particular element
     HINT: You have to use the compare() function to compare values.
     param: tree    the binary search tree
     val        the value to search for in the tree
     pre:   tree is not null
     val is not null
     post:  none

         ***************************************************************************
         **************/
```

```
int containsBSTree(struct BSTree *tree, TYPE val) {
    assert(tree != NULL);
    assert(val != NULL);

    /*write this*/
    struct Node * thisNode;
    thisNode = tree->root;

    while (thisNode != NULL) {
        if (compare(val, thisNode->val) == 0)       //found it
            return 1;
        else if (compare(val, thisNode->val) == -1)     //d < this node
            thisNode = thisNode->left;
        else if (compare(val, thisNode->val) == 1)      //d > this node
            thisNode = thisNode->right;
    }
    return 0;                           //failed to find it
}

/
    ***************************************************************************
    **************
 helper function to find the left most child of a node
 return the value of the left most child of cur
 param: cur     the current node
 pre:   cur is not null
 post: none

     ***************************************************************************
     **************/
TYPE _leftMost(struct Node *cur)
{
    /*write this*/
    while (cur->left != NULL) {
        cur = cur->left;    //move to left subtree until null is encountered
    }
    return cur->val;            //return value at current [leftmost] node

}


/
    ***************************************************************************
    **************
 recursive helper function to remove the left most child of a node
 HINT: this function returns cur if its left child is NOT NULL. Otherwise,
 it returns the right child of cur and free cur.

 Note:  If you do this iteratively, the above hint does not apply.

 param: cur the current node
 pre:   cur is not null
 post:  the left most node of cur is not in the tree

     ***************************************************************************
     **************/
```

```
struct Node *_removeLeftMost(struct Node *cur) {
    /*write this*/
    struct Node *temp;


    //baseCase
    if (cur->left == NULL) {            //curent node == leftmost
            temp = cur->right;              //save right descendant(s) if any
            free(cur);                      //free leftMost
            return temp;                    //return the right child to the
                caller
    }
    //recursive case; continue traversing to leftMost
    else
        cur->left = _removeLeftMost(cur->left ); //move further down
    return cur;             //return current (moded) node to caller to rebuild
        the tree
}
/
    ***************************************************************************
    **************
 recursive helper function to remove a node from the tree
 HINT: You have to use the compare() function to compare values.
 param: cur the current node
 val    the value to be removed from the tree
 pre:   val is in the tree
 cur is not null
 val is not null

     ***************************************************************************
     **************/
struct Node *_removeNode(struct Node *cur, TYPE val) {
    /*write this*/
    assert (cur != NULL);
    struct Node *temp;

    //baseCase
    if (compare(val, cur->val) == 0) {      //found it
        if (cur->right == NULL) {
            temp = cur->left;
            free(cur);                      //freed it
            return temp;                    //return left child
        }
        else {
            cur->val = _leftMost(cur);
            cur->right = _removeLeftMost(cur);
        }

    }


    //recursive case; keep searching for value d
    if (compare(val, cur->val) == -1)
        cur->left = _removeNode (cur->left , val);   //d is less than current
    else if (compare(val, cur->val) == 1)
        cur->right = _removeNode (cur->right, val);  //d is >= current
    return cur;            //return current (moded) node to caller to rebuild
        the tree
}



/
    ***************************************************************************
    **************
 function to remove a value from the binary search tree
 param: tree   the binary search tree
 val        the value to be removed from the tree
 pre:   tree is not null
 val is not null
 val is in the tree
 pose:  tree size is reduced by 1

     ***************************************************************************
     **************/
void removeBSTree(struct BSTree *tree, TYPE val)
{
    if (containsBSTree(tree, val)) {
        tree->root = _removeNode(tree->root, val);
        tree->cnt--;
    }
}

/
    ***************************************************************************
    **************

     ***************************************************************************
     ***************

     ***************************************************************************
     ***************

     ***************************************************************************
     ***************

     ***************************************************************************
     **************/

#if 1
#include <stdio.h>

/*------------------------------------------------------------------------------
    */
void printNode(struct Node *cur) {
    if (cur == 0) return;
    printf("(");
    printNode(cur->left);
    /*Call print_type which prints the value of the TYPE*/
    print_type(cur->val);
    printNode(cur->right);
    printf(")");
```

```
}

void printTree(struct BSTree *tree) {
    if (tree == 0) return;
    printNode(tree->root);
}
/*----------------------------------------------------------------------------
    */

#endif

/
    ****************************************************************************
    **************
 *********************************
     *********************************
 ********************************     TESTING FUNCTIONS
     ********************************
 *********************************
     *********************************

     ***************************************************************************
     **************/

#if 1
/
    ****************************************************************************
    ***************
 function to build a Binary Search Tree (BST) by adding numbers in this
     specific order
 the graph is empty to start: 50, 13, 110 , 10

     ***************************************************************************
     ***************/
struct BSTree *buildBSTTree() {
    /*     50
     13  110
     10
     */
    struct BSTree *tree = newBSTree();

    /*Create value of the type of data that you want to store*/
    struct data *myData1 = (struct data *) malloc(sizeof(struct data));
    struct data *myData2 = (struct data *) malloc(sizeof(struct data));
    struct data *myData3 = (struct data *) malloc(sizeof(struct data));
    struct data *myData4 = (struct data *) malloc(sizeof(struct data));

    myData1->number = 50;
    myData1->name = "rooty";
    myData2->number = 13;
    myData2->name = "lefty";
    myData3->number = 110;
    myData3->name = "righty";
    myData4->number = 10;
    myData4->name = "lefty of lefty";
    /*add the values to BST*/
    addBSTree(tree, myData1);
    addBSTree(tree, myData2);
    addBSTree(tree, myData3);
    addBSTree(tree, myData4);

    return tree;
}

/
    ***************************************************************************
    **************
 function to print the result of a test function
 param: predicate:  the result of the test
 nameTestFunction : the name of the function that has been tested
 message

     ***************************************************************************
     **************/
void printTestResult(int predicate, char *nameTestFunction, char *message){
    if (predicate)
        printf("%s(): PASS %s\n",nameTestFunction, message);
    else
        printf("%s(): FAIL %s\n",nameTestFunction, message);
}

/
    ***************************************************************************
    **************
 fucntion to test each node of the BST and their children //4 allocs

     ***************************************************************************
     **************/
void testAddNode() {
    struct BSTree *tree = newBSTree();

    struct data myData1,  myData2,  myData3,  myData4;

    myData1.number = 50;
    myData1.name = "rooty";
    addBSTree(tree, &myData1);
    //check the root node
    if (compare(tree->root->val, (TYPE *) &myData1) != 0) {
        printf("addNode() test: FAIL to insert 50 as root\n");
        return;
    }
    //check the tree->cnt value after adding a node to the tree
    else if (tree->cnt != 1) {
        printf("addNode() test: FAIL to increase count when inserting 50 as
            root\n");
        return;
    }
    else printf("addNode() test: PASS when adding 50 as root\n");


    myData2.number = 13;
    myData2.name = "lefty";
    addBSTree(tree, &myData2);

    //check the position of the second element that is added to the BST tree
    if (compare(tree->root->left->val, (TYPE *) &myData2) != 0) {
        printf("addNode() test: FAIL to insert 13 as left child of root\n");
        return;
    }
    else if (tree->cnt != 2) {
        printf("addNode() test: FAIL to increase count when inserting 13 as
            left of root\n");
        return;
    }
    else printf("addNode() test: PASS when adding 13 as left of root\n");


    myData3.number = 110;
    myData3.name = "righty";
    addBSTree(tree, &myData3);

    //check the position of the third element that is added to the BST tree
    if (compare(tree->root->right->val, (TYPE *) &myData3) != 0) {
        printf("addNode() test: FAIL to insert 110 as right child of root\n");
        return;
    }
    else if (tree->cnt != 3) {
        printf("addNode() test: FAIL to increase count when inserting 110 as
            right of root\n");
        return;
    }
    else printf("addNode() test: PASS when adding 110 as right of root\n");


    myData4.number = 10;
    myData4.name = "righty of lefty";
    addBSTree(tree, &myData4);

    //check the position of the fourth element that is added to the BST tree
    if (compare(tree->root->left->left->val, (TYPE *) &myData4) != 0) {
        printf("addNode() test: FAIL to insert 10 as left child of left of
            root\n");
        return;
    }
    else if (tree->cnt != 4) {
        printf("addNode() test: FAIL to increase count when inserting 10 as
            left of left of root\n");
        return;
    }
    else printf("addNode() test: PASS when adding 10 as left of left of root\n"
        );
}

/
    ****************************************************************************
    **************
 fucntion to test that the BST contains the elements that we added to it // 4
```

```
     allocs


     ***************************************************************************
     **************/
void testContainsBSTree() {
    struct BSTree *tree = buildBSTTree();

    struct data myData1,  myData2,  myData3,  myData4, myData5;

    myData1.number = 50;
    myData1.name = "rooty";
    myData2.number = 13;
    myData2.name = "lefty";
    myData3.number = 110;
    myData3.name = "righty";
    myData4.number = 10;
    myData4.name = "lefty of lefty";
    myData5.number = 111;
    myData5.name = "not in tree";

    printTestResult(containsBSTree(tree, &myData1), "containsBSTree", "when
        test containing 50 as root");

    printTestResult(containsBSTree(tree, &myData2), "containsBSTree", "when
        test containing 13 as left of root");

    printTestResult(containsBSTree(tree, &myData3), "containsBSTree", "when
        test containing 110 as right of root");

    printTestResult(containsBSTree(tree, &myData4), "containsBSTree", "when
        test containing 10 as left of left of root");

    //check containsBSTree fucntion when the tree does not contain a node
    printTestResult(!containsBSTree(tree, &myData5), "containsBSTree", "when
        test containing 111, which is not in the tree");

}

/
    ***************************************************************************
    **************
 fucntions to test the left_Most_element

     ***************************************************************************
     **************/
void testLeftMost() {
    struct BSTree *tree = buildBSTTree();

    struct data myData3, myData4;

    myData3.number = 110;
    myData3.name = "righty";
    myData4.number = 10;
    myData4.name = "lefty of lefty";
```

```
        printTestResult(compare(_leftMost(tree->root), &myData4) == 0, "_leftMost",
            "left most of root");

        printTestResult(compare(_leftMost(tree->root->left), &myData4) == 0,
            "_leftMost", "left most of left of root");

        printTestResult(compare(_leftMost(tree->root->left->left), &myData4) == 0,
            "_leftMost", "left most of left of left of root");

        printTestResult(compare(_leftMost(tree->root->right), &myData3) == 0,
            "_leftMost", "left most of right of root");

    }

    void testRemoveLeftMost() {
        struct BSTree *tree = buildBSTTree();
        struct Node *cur;

        cur = _removeLeftMost(tree->root);

        printTestResult(cur == tree->root, "_removeLeftMost", "removing leftmost of
            root 1st try");

        cur = _removeLeftMost(tree->root->right);
        printTestResult(cur == NULL, "_removeLeftMost", "removing leftmost of right
            of root 1st try");

        cur = _removeLeftMost(tree->root);
        printTestResult(cur == tree->root, "_removeLeftMost", "removing leftmost of
            root 2st try");
    }

    void testRemoveNode() {
        struct BSTree *tree = buildBSTTree();
        struct Node *cur;
        struct data myData1,  myData2,  myData3,  myData4;

        myData1.number = 50;
        myData1.name = "rooty";
        myData2.number = 13;
        myData2.name = "lefty";
        myData3.number = 110;
        myData3.name = "righty";
        myData4.number = 10;
        myData4.name = "lefty of lefty";

        _removeNode(tree->root, &myData4);
        printTestResult(compare(tree->root->val, &myData1) == 0 && tree->root->left
            ->left == NULL, "_removeNode", "remove left of left of root 1st try");

        _removeNode(tree->root, &myData3);
        printTestResult(compare(tree->root->val, &myData1) == 0 && tree->root->
            right == NULL, "_removeNode", "remove right of root 2nd try");

        _removeNode(tree->root, &myData2);
        printTestResult(compare(tree->root->val, &myData1) == 0 && tree->root->left
            == 0, "_removeNode", "remove right of root 3rd try");

        cur = _removeNode(tree->root, &myData1);
        printTestResult(cur == NULL, "_removeNode", "remove right of root 4th try")
            ;
    }
    /
        ****************************************************************************
        **************
     *********************************
         *********************************
     *********************************    MAIN() FUNCTION
         *********************************
     *********************************
         *********************************

         ****************************************************************************
         **************/
    /*

     Main function for testing different fucntions of the Assignment#4.

     */

    int main(int argc, char *argv[]){

        //After implementing your code, please uncommnet the following calls
        //to the test functions and test your code

         testAddNode();

        printf("\n");
            testContainsBSTree(); //4 allocs 0 freed

        printf("\n");
            testLeftMost(); //4allocs 0 frees

        printf("\n");
            testRemoveLeftMost();

        printf("\n");
            testRemoveNode(); // 4 allocs


        return 0;


    }

    #endif
```