CSE312-L: DSA
LNMIIT, Rupa-ki-Nangal, Jaipur 302031
Training Set 06

Training sets 06 guides the students to practice writing ADT interface functions related to a binary search tree described in course textbook (R Thareja: Data Structures Using C, 2nd edn, Oxford University Press, 2014). Students will also use the data-structure to complete an application.

The ADT functions of interest to us in this training set are best listed through file **`srchTree.h`**:

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

struct node {
     int key;
     int height;
     int depth;
     struct node *left;
     struct node *right;
};

void initTree();
void insertKey(int key);
void deleteKey(int key);
int hasKey(int key);
void setDepths();
void setHeights();
void printTree();
```

These functions are used in program **`main.c`** printed below:

```
#include "srchTree.h"

int main(void) {
     initTree();
     insertKey(14); insertKey(17); insertKey(11);
     insertKey(17); insertKey(7); insertKey(53); insertKey(4);
     insertKey(13); insertKey(12); insertKey(8); insertKey(60);

     printf("\n----------- INSERTIONS COMPLETED ------------\n");
     if (hasKey(17))
          printf("Found key 17\n");
     else
          printf("Not Found key 17\n");
```

```
      printTree();
      printf("\n--------------- BEGIN DELETIONS ------------\n");

      deleteKey(60);
      deleteKey(53);
      deleteKey(17);
      if (hasKey(17))
            printf("Found key 17\n");
      else
            printf("Not Found key 17\n");
      deleteKey(14);
      printf("\n------------ DELETIONS COMPLETED --------------\n");
      printTree();

      return 0;
}
```

To support your understanding, the training set lists the output of this program.

```
------------- INSERTIONS COMPLETED ---------------
Found key 17

                                    60
                          53
                17
START -> 14
                          13
                                    12
                11
                                    8
                          7
                                    4

--------------- BEGIN DELETIONS ------------
Not Found key 17

------------ DELETIONS COMPLETED --------------

                13
                          12
START -> 11
                          8
                7
                          4
```

The printout displays two somewhat rough views of the tree at two separate points during a run of the program. Though rough, the display helps us see the tree and its nodes clearly. The primary benefit of the simplicity will be ease of coding; one of the tasks will require the students to print this tree.

At the end of this document there is another version of displayed tree that marks each key value with the letter L or R to indicate if the value is from a left child of a right child. It can be done by passing an additional char parameter to functions `printWell()` and `printNode()`.

To support the students in their efforts, implementations of the following ADT interface functions is provided in the code listed later in this document. A close attention and study of these functions will help you code the other functions.

```
void initTree();
int hasKey(int key);
void setDepths();
void setHeights();
```

The students will be asked to code and test the following ADT interface functions:

```
void insertKey(int key);
void deleteKey(int key);
void printTree();
```

Task 01:

Task 01 of the exercise requires the students to correctly implement ADT function:

```
void insertKey(int key);
```

As you note in the code below, the function is implemented using a static function in file `srchTree.c`. A static function is only accessible to the functions in the file. Try calling a static file in function `main()`. You will get an error message from your compiler and linker `gcc`.

You will need to implement function:

```
static struct node *insert(int key, struct node *tree)
```

This is a recursive function of fewer than 15 lines. Much of the code is already provided in this document. Test your program by using functions already provided in the code.

Task 02:

In task 02, students are required to write code for static function `void printWell(struct node *tree, int spaces)`

This is again less than 10 lines of code and is primarily based on in-order search of the tree. This is a recursive function that increases the amount of white spaces by amount equal to a STEP as it descends over the nodes of the tree.

On completion of this program you should be able to display the tree as shown previously. I hope you might have noticed that that classical approach of visiting left tree first produces a mirror image of the trees shown above. You may reverse the visit order to alter the placements of the nodes.

Task 03:

Writing code for ADT interface function `deleteKey()` requires more sophistication and care. As you note the code available to the students implements it through three functions. There are many approaches to deleting a key from a search tree.

The one, I have used in my coding is as follows:

1. We have a recursive function `static struct node * deleteNode(int key, struct node *tree)` that seeks to delete the node with `key`. The caller must be prepared to accept a changed tree from the call. The root of the subtree may be modified or even deleted as a result of the call.
2. Once a node with key is located, it will be deleted. This deletion may occur in one of the following scenarios:
    a. The node with the key has no subtree below it. In this case the node is deleted and the caller must replace the node with a NULL (empty) tree.
    b. The node with the matching key has only one subtree. This can be either a left subtree or a right subtree. In these cases, the deleted node can be substituted by the root of the non-empty subtree.
    c. The last scenario is about the node that matches the key and has both subtrees. In this case, it is easy to find a new key to take place of the deleted key. In the code listed, function `static int graftReplacementKey(struct node *tree)` returns this key. This key is immediate predecessor of the key being deleted and found in the left subtree attached to the node with key being removed. The support function also removes the node which provides the replacement key. It should be obvious that the caller must make sure (before the call) that the node that `graftReplacementKey()` removes is not the one directly being pointed to by the node with the key to be removed.

If student needs further reference, they may consult the textbook or the class notes. The textbook has implemented a similar algorithm using an iterative process.

srchTree.c

```
#include "srchTree.h"
#include <assert.h>
#include <string.h>

#define SCRN_WIDTH 100
#define OFFSET 9

struct node *theTree;
int STEP;

static struct node *makeNode(int key) {
     struct node *new = malloc(sizeof(struct node));
     assert(new != NULL);
     new->left = new->right = NULL;
     new->key = key;
     new->height = 0;
     new->depth = 0;
     return new;
}

void initTree() {
     theTree = NULL;
}

static int find(int key, struct node *t) {
     if (t == NULL)
          return 0;
     if (t->key == key)
          return 1;
     if (t->key > key)
          return find(key, t->left);
     else
          return find(key, t->right);
}

int hasKey(int key) {
     return find(key, theTree);
}
```

```
static struct node *insert(int key, struct node *tree) {
     struct node *t;
     if (tree == NULL) {
          t = makeNode(key);
          return t;
     }
/* TASK 01
     ONLY A SHORT CODE REMOVED
*/
}

void insertKey(int key) {
     theTree = insert(key, theTree);
}

static int setNodeHeights(struct node *tree) {
     int lh, rh;
     if (tree == NULL)
          return 0;
     lh = setNodeHeights(tree->left)+1;
     rh = setNodeHeights(tree->right)+1;
     tree->height = lh>rh?lh:rh;
     return tree->height;
}

void setHeights() {
     theTree->height = setNodeHeights(theTree);
}

static void setNodeDepths(struct node *tree, int depth) {
     if (tree == NULL)
          return;
     tree->depth = depth+1;
     setNodeDepths(tree->left, tree->depth);
     setNodeDepths(tree->right, tree->depth);
}

void setDepths() {
     setNodeDepths(theTree, 0);
}
```

```
static int graftReplacementKey(struct node *tree) {
     struct node *parent;
     int replacementKey;
     assert(tree != NULL);
     while (tree->right != NULL) {
          parent = tree;
          tree = tree->right;
     }
     replacementKey = tree->key;
     parent->right = tree->left;
     free(tree);
     return replacementKey;
}


static struct node * deleteNode(int key, struct node *tree) {
     struct node * tmp;
     int replacementKey;

     if (tree == NULL)
          return tree;

     if (tree->key > key) {
          tree->left = deleteNode(key, tree->left);
          return tree;
     } else if (tree->key < key) {
          tree->right = deleteNode(key, tree->right);
          return tree;
     }

     /* The remaining case when node is deleted */
     assert(tree->key == key);
/* TASK 03
     ABOUT 30 LINES OF CODE REMOVED
*/
     tree->key = graftReplacementKey(tree->left);
     return tree;
}

void deleteKey(int key) {
     theTree = deleteNode(key, theTree);
}
```

```
static void printNode(struct node *tree, int spaces) {
     if (tree == NULL) {
          //printf("\n");
          return;
     }

     while (spaces-->0)
          printf(" ");
     printf("%d\n", tree->key);
}


static void printWell(struct node *tree, int spaces) {
     if (tree == NULL) {
          printNode(NULL, spaces);
          return;
     }
/* TASK 02
     A SHORT SEQUENCE OF CODE REMOVED
*/
}

void printTree() {
     int i;
     if (theTree == NULL) {
          printf("START----> NULL\n");
          return;
     }

     setHeights();
     setDepths();
     STEP = SCRN_WIDTH/theTree->height;
     if (STEP>9)
          STEP = 9;
     printf("\n");
     printWell(theTree->right, OFFSET+STEP);
     printf("START -> %d\n", theTree->key);
     printWell(theTree->left, OFFSET+STEP);
}
```

```
                    53R
               12R
                    11L
START -> 10root
               9L
                    8L
                         7L
                              6L
                                   5L
                                             4R
                                        3L
```