

```
#ifndef __BINARY_TREES_
#define __BINARY_TREES_

/* vlad: data types */
typedef enum {FALSE, TRUE} BOOL;

typedef struct Nod
{
    int inf;           /* vlad: informatia */
    struct Nod *leg_st, *leg_dr; /* vlad: referinte descendente */
} NOD, *pNOD; /* vlad: very similar with linked-lists */

/* vlad: functions' prototypes */
void ins_nod(pNOD*, int);
BOOL del_nod(pNOD*, int);
int tree_height( pNOD*, int*, int* );
BOOL isBalanced( pNOD*, int, int );

void inOrder(pNOD);
void preOrder(pNOD);
void postOrder(pNOD);

#endif
```

```

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

#include "binary_trees.h"

/* vlad: insertion and deletion functions */

/* programul va primi ca parametru valoarea nodului pe care trebuie sa îl adauge.
 * vlad: Algoritmul seamana cu 'sortarea prin interclasare': gasesc pozitia
 * (locul) potrivita unde urmeaza sa fac inserarea si stochez nodul
 * in acea pozitie.
 * Surse bibliografice:
 * - Niklaus Wirth - Algo + data structures = programs
 * - http://software.ucv.ro/~cstoica/ISP/Lucrarea%209%20-%20arbori.pdf (!)
 * - http://infoscience.3x.ro/c++/stergere_abc.htm
 * - http://staff.cs.upt.ro/~chirila/teaching/upt/id21-aa/lectures/AA-ID-Cap08-
 * 2.pdf (!)
 * - http://andrei.clubcisco.ro/cursuri/1sd/curs/curs06.pdf
 */

void ins_nod(pNOD *pPrim, int val)
{
    pNOD nod1, nod2, nod3; /* vlad: reference types, since I have to use the links */
    /* se creaza nodul care se adauga în arbore. */
    nod1 = (pNOD) malloc(sizeof(NOD));
    assert(nod1 != NULL);

    nod1->inf = val;           /* vlad: information stored in the node */

    /* deoarece o sa se adauge ca nod frunza (!) => nu va avea nici un copil,
     * deci legaturile stanga si dreapta vor fi nule.
     * The following two lines define a leaf-node
     */
    nod1->leg_st = NULL;
    nod1->leg_dr = NULL;

    /* verificam mai intai daca exista o radacina (daca arborele a fost creat) */
    if(*pPrim == NULL)
    {
        /* nu am mai introdus nimic pana acum =>
         * nodul creat va deveni radacina arborelui
         */
        *pPrim = nod1;
    } else {          /* Daca exista un nod radacina, inserarea în arbore
                      * se va face conform algoritmului prezentat la curs.
                      */
        nod2 = *pPrim;      /* vlad: plec din radacina */
        /* se va cauta locul de inserare al nodului 'nod1',
         * pornind cautarea din radacina.
         * Nodul 'nod3' va reprezenta nodul parinte al nodului 'nod1'
         */
        do {

```

```

        /* vlad: avansez la stanga (daca: val < val_nod) */
        if (val < nod2->inf) {
            nod3 = nod2; /* vlad: pastrez ultima legatura
                           * inainte de NULL
                           */
            nod2 = nod2->leg_st;
        } else /* se merge spre dreapta (daca: val_de_inserat > val_nod) */
        {
            nod3 = nod2;
            nod2 = nod2->leg_dr;
        }/* end else */
    } while (nod2 != NULL); /* end do-while */

    /* dupa gasirea nodului parinte, se creaza legatura
     * nod3 (nod parinte) -> nod1 (nodul inserat acum)
     */
    if (val < nod3->inf) /* vlad: should NOT be nod2 here... */
        /* se asaza in stanga parintelui */
        nod3->leg_st = nod1;
    else
        /* se asaza in dreapta parintelui */
        nod3->leg_dr = nod1;
}/* end else */

return;
}

/* vlad: sterge nod si pastreaza caracteristicile unui arbore binar de cautare
 * Pentru ca arborele sa-si pastreze proprietatile, la stergere trebuie avute in
vedere 3 situatii:
 * Cazul I: nodul de sters NU are fiu stanga;
 * Cazul II: nodul de sters NU are fiu dreapta;
 * Cazul III: nodul de sters are ambii fii.
 *
 * Principiul este de a inlocui nodul sters cu 'valoarea nodului celui mai din
stanga, al subarborelui sau drept'. Adica nodul cel mai mic din subarborele drept.
 * Tot un arbore binar de cautare este si atunci cand inlocuiesc nodul de sters cu
valoarea maxima din subarborele stang.
 */
BOOL del_nod(pNOD *pPrim, int nr)
{
    /*BOOL del_root = FALSE;*/
    pNOD tmp, tmp1, tmp2, min;

    /* se porneste cautarea informatiei de sters din radacina */
    assert(*pPrim != NULL); /* vlad: nu incerc sa parcurg un arbore vid =>
                           * eroare dc primesc NULL!
                           */
    tmp1 = *pPrim; /* vlad: caut valoarea de sters; incep cu radacina */
    do {
        tmp = tmp1;
        if (tmp1->inf > nr)
            tmp1 = tmp1->leg_st; /* vlad: avansez pe subarbore stang */
        else if (tmp1->inf < nr )

```

```

        tmp1 = tmp1->leg_dr;      /* vlad: avansez pe subarbore drept */
        /* vlad: valoarea este in nodul radacina */
    } while(tmp1 && tmp1->inf != nr);

assert(tmp != NULL);
if(tmp == NULL) return FALSE;      /* vlad: no such node in the tree! */

if( (tmp1->leg_dr == NULL) && (tmp1->leg_st == NULL) )      /* vlad: nod frunza */
{
    fprintf(stdout, "Nod frunza\n");
    if(tmp->leg_dr == tmp1) tmp->leg_dr = tmp1->leg_dr;
    if(tmp->leg_st == tmp1) tmp->leg_st = tmp1->leg_st;
    free(tmp1);
} else if( tmp1->leg_dr && tmp1->leg_st )      /* cazul III: amandoi fiii */
{
    /* mergem la copilul din dreapta: subarborele drept */
    min = tmp1->leg_dr;      /* vlad: primul nod al subarborelui drept */
    tmp2 = tmp1; /* vlad: what if I don't enter the while below?
                  * Well, this statement will be valid afterward. I can
                  * take
                  */
    /* cautam cel mai din stanga copil (din subarborele drept)
     * Cum ziceam mai sus, o varianta echivalenta este: maximul din
     * subarborele stang.
     * Se poate alege intre oricare din cele doua variante: structura de
     * BST se pastreaza.
     */
    while(min->leg_st)
    {
        tmp2 = min; /* vlad: retin tatal nodului celui mai din stanga */
        min = min->leg_st;      /* vlad: nodul cel mai din stanga: nu
                                    * are subarbore stang, are doar drept
                                    */
    }

    /* vlad: min este parintele ultimului nod frunza;
     * tmp2 este parintele lui min
     */
    assert(min != NULL);
    tmp1->inf = min->inf;
    fprintf(stdout, "Noua informatie a nodului: %i\n", tmp1->inf);
    tmp2->leg_dr = min->leg_dr;
        /* vlad: this is the most important note here!
         * I don't alter the left subtree (which I preserve).
         * Since I've followed the right subtree, then
         * the left one should not be touched.
         * I will only alter the right one - and it disappears,
         * here.
         */
    fprintf(stdout, "Cazul 3: ambii fii -> am eliberat nodul cerut \n");
    free(min);      /* vlad: eliberarea zonei de memorie */
} /* end: cazul III */

else if ( tmp1->leg_st )      /* cazul I: am doar fiu stanga */

```

```

{
    pNOD fiu = NULL;

    fiu = tmp1->leg_st;
    tmp1->inf = fiu->inf; /* mutare inf din nodul din st, in nodul curent */
    tmp1->leg_st= fiu->leg_st; /* refacere legaturi */

    fprintf(stdout, "Cazul 1: doar fiu stanga -> am eliberat nodul
                           cerut\n");
    free(fiu); /* vlad: eliberarea zonei de memorie */
} else /* cazul II: am doar fiu dreapta */
{
    pNOD fiu = NULL;

    fiu = tmp1->leg_dr;
    /* mutare inf din nodul din st, in nodul curent */
    tmp1->inf = fiu->inf;
    tmp1->leg_dr= fiu->leg_dr;
    fprintf(stdout, "Cazul 2: doar fiu dreapta -> am eliberat
                           nodul cerut\n");
    free(fiu); /* vlad: eliberarea zonei de memorie */
}
return TRUE;
}

```

```

/* vlad: parcurgeri */
void inOrder(pNOD pRad)
{
    /* daca nu s-a ajuns la ultimul nod */
    if (pRad != NULL)
    {
        /* se viziteaza copilul din stanga */
        inOrder(pRad->leg_st);

        /* se viziteaza radacina (aici la mijloc) */
        printf(stdout, "%d -> ", pRad->inf);

        /* se viziteaza copilul din dreapta */
        inOrder(pRad->leg_dr);
    }
    return;
}

void preOrder(pNOD pRad)
{
    /* daca nu am arbore vid (la intrare) SI
     * daca nu s-a ajuns la ultimul nod (in timpul rularii)
     */
    if (pRad != NULL)
    {
        /* se viziteaza radacina (aici prima) */
        printf(stdout, "%d -> ", pRad->inf);

```

```

/* se viziteaza copilul din stanga, apoi cel din dreapta */
preOrder(pRad->leg_st);
preOrder(pRad->leg_dr);
}

return;
}

void postOrder(pNOD pRad)
{
    /* daca nu s-a ajuns la ultimul nod */
    if (pRad != NULL)
    {
        /* se viziteaza copilul din stanga, apoi cel din dreapta
         * (this is the mandatory order)
         */
        postOrder(pRad->leg_st);
        postOrder(pRad->leg_dr);

        /* se viziteaza radacina (aici ultima) */
        fprintf(stdout, "%d -> ", pRad->inf);
    }
    return;
}

/* vlad: compute the height of a given node (if the input node
 * is the root => compute the tree's height
 * The height essentially depends upon >the order< the input keys
 * are fed into this program (!)
 */
int tree_height(pNOD *pRoot, int *pHL, int *pHR)
{
    if(*pRoot == NULL )
        return 0; /* vlad: convention says that, for a void tree,
                   * its height is -1 (in fact, non-existent)
                   */
    /* vlad: the computational relation is:
     * height = 1 + max{height(root.left), height(root.right)}
     */
    return 1 + ( (*pHL = tree_height( &(*pRoot)->leg_st, pHL, pHR ) > (*pHR =
tree_height( &(*pRoot)->leg_dr, pHL, pHR)) ? *pHL : *pHR);
}

BOOL isBalanced(pNOD *pRoot, int HL, int HR)
{
    return
        (      abs(HL-HR) <= 1 &&
              isBalanced(&(*pRoot)->leg_st, HL, HR) &&
              isBalanced(&(*pRoot)->leg_dr, HL, HR));
}

```

```

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

#include "binary_trees.h"

int main()
{
    int i, n, val;
    int left_height, right_height;
    BOOL ret_delete;

    pNOD r = NULL;           /* vlad: the root;
                                * At the very beginning time the tree is empty (!)
                                * I will assume it this way.
                                */
}

/* building the tree */
fprintf(stdout, "nr. de noduri: \n"); scanf("%i", &n);
assert(n > 0);

for(i=1; i<=n; i++)
{
    fprintf(stdout, "valoarea de inserat: "); scanf("%i", &val);
    ins_nod(&r, val); /* vlad: each node is a leaf */
}

/* tree traversals */
puts("");
fprintf(stdout, " Arborele contine valorile: (inorder)\n");
inOrder(r);

puts("");
fprintf(stdout, " Arborele contine valorile: (preorder)\n");
preOrder(r);

puts("");
fprintf(stdout, "Arborele contine valorile: (postorder)\n");
postOrder(r);

puts("");           /* newline, given the actual display style
                      * (I have no newline between the traversals
                      */
left_height = right_height = 0;
fprintf(stdout, "The tree's height is (before deletion): %d\n",
        tree_height(&r, &left_height, &right_height));
fprintf(stdout, "Left height: %i, right height: %i\n",
        left_height, right_height);

/* is it a balanced tree? */
puts("");
fprintf(stdout, "Is balanced? %s \n",
        isBalanced(&r, left_height, right_height)?"Yes":"No");

```

```

/* tree deletions (not all nodes) */
for(i=1; i<n; i++)
{
    puts("");
    do
    {
        fprintf(stdout, "valoarea de sters "); scanf("%i", &val);
        if( (ret_delete = del_nod(&r, val)) == FALSE )
            fprintf(stdout, "Nod inexistent! Nu pot sterge...\n");
    } while( ret_delete == FALSE );

    fprintf(stdout, "Dupa stergere (inorder):\n");
    inOrder(r);

    /* vlad: compute tree's height after deletion */
    puts("");
    left_height = right_height = 0;
    fprintf(stdout, "The tree's height is (after deletion): %d\n",
            tree_height(&r, &left_height, &right_height));
    fprintf(stdout, "Left height: %i, right height: %i\n",
            left_height, right_height);
}

fflush(stdin);
return 0;
}

```