

```

/* The (personal) header file for this project (lsi.h)
 * Declarations, type definitions, functions prototypes etc.
 */
#ifndef __LIST_EXAMPLE__
#define __LIST_EXAMPLE__

typedef unsigned long int ULI;           // a new data type

typedef struct Persoana
{
    char Nume[64];
    ULI length;
    struct Persoana *pNext; /* vlad: only one link here: to the next node */
} NODE, *pNODE, **ppNODE; /* The new user defined data types, useful here */

typedef enum boolean{False, True} BOOL; /* Note: BOOL != unsigned int */

void addNode(ppNODE, pNODE, char*); /* vlad: the convention made for the 2nd
                                         * argument here is to preserve the link (pointer)
                                         * to the node preceding (before) the one to be inserted.
                                         */
void delNode(ppNODE, pNODE); /* vlad: the convention made here for the 2nd
                                * argument tells us that we preserve the pointer to
                                * some node, the one before the deleted one.
                                */
void listTraversal(pNODE, unsigned long int*); /* vlad: start with the 1st node
                                                 * and move onward.
                                                 */
pNODE searchUltimate(pNODE); /* vlad: adaptation of a traversal in order to
                                * return a node's address */
BOOL searchValue(pNODE, char*); /* vlad: adaptation of a traversal in order to
                                * find a value */

BOOL isEmpty(pNODE); /* vlad: this checks if the list is empty */

#endif

```

```

/* vlad: this is the very header file for Hash-tables (hash_tbls.h)
 * It contains solely the interface declarations for hash-tables
 * Note: for LSI I have to add the files separately (within this project)
 */

#ifndef __HASH_TBLS__
#define __HASH_TBLS__

#include "lsi.h"

#define DIM 26
#define MASK 0xffffffff // ignore the sign bit (although I'm already using
                     // natural 32b values)
                     // 0111 1111 1111 1111 1111 1111 1111 1111 (binary)
#define DEBUG // enables the debugging messages (here they are mostly fprintfs)

// vlad: for the hashing function, once I know the index, then the time to
// look-up any value in the array is >independent< both of the size of that array,
// and the position of the required data in that array
// The hashing function (hash algorithm) transforms (translates) the input data
// into indices.
// Those indices should NOT overflow the maximum # of indices I have in mind (I'm
// effectively using in a particular situation).
// hash(key) => index

typedef unsigned long int ULI;           // 32b positive integers (4B)
typedef unsigned long long int ULLI;      // 64b positive integers (8B)

ULLI hash_fct(ULLI, ULI, char*); // use the hash function dedicated to strings
ULI ins-HT(pNODE[], char*, ULLI, ULI); // parameters: the hash-table and the
                                         // input information:
                                         // - value of m (large prime number)
                                         // - value of p (prime number)
void listTrav-HT(pNODE); // a list traversal variant, dedicated to hash-
                           // tables (has no length as argument)
BOOL lookout-HT(pNODE[], char*, ULLI, ULI);
  // the parameters are:
  // - the hash table (the linear collection of prim nodes for each linked list)
  // - the string
  // - the value of m
  // - the value of p

void inspect-HT(pNODE[]); // parse through the table and display the
                           // information already stored

#endif

```

```

/* The translation unit dedicated to the implementations of functions (lsi.c)
 * This is another module within this project.
 * This is where all the implementations of our functions should belong.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "lsi.h"           /* vlad: the new designed header file is required here.
                           * Otherwise the functions below are not known at all.
                           */
void addNode(ppNODE ppPrim, pNODE p, char *Name) /* vlad: we insert anywhere in
                                                 * the list, but AFTER 'p'(!) */
/* vlad: from arguments' perspective, a function call is by default
 * performed 'by value'. This also stands for addresses!
 * Here the address of the first node in the list is passed to the function.
 * The changes within this function may relate to this address. Therefore,
 * they are required outside of this function as well...
 * That's why I should have a double pointer for prim (ppPrim = &prim)
 * so that I can see when it changes from outside the function!
 */
{
    pNODE q = (pNODE) malloc(sizeof(NODE));
    assert(q != NULL);
    fprintf(stdout, "New node's address: %p\n", q);

    /* The information (data) of the new node. It should be asked each time */
    q->anNastere = 2001;
    /* vlad: how can we automate the process here (and below)?
     * Some inter-calls data must be preserved...
     */
    strcpy(q->Nume, Name);
    q->Nume[strlen(q->Nume)] = '\0';

    /* The links of the new node */
    if(p == NULL) /* vlad: insert at the beginning */
    {
        q->pNext = *ppPrim; /* At the very first call of addNode(),
                               * the list has only one node!
                               * It doesn't point to anything after it!
                               */
        *ppPrim = q;          /* assume that prim is now NOT NULL */
        fprintf(stdout, "prim's address: %p\n", *ppPrim);
    } else { /* vlad: insert anywhere in the list, after p (!) */
        q->pNext = p->pNext; /* Preserve the link, in the first place... */
        p->pNext = q;         /* ... after which you can alter it.
                               * The new node (here, q) clearly comes after p.
                               */
    }
    return;
}

```

```

void delNode(ppNODE ppPrim, pNODE p)
{
    pNODE q;
    if( p == NULL )      /* vlad: we're about to delete the 1st node */
    {
        q = *ppPrim;      /* q retains the address of the node to be deleted.
                           * Here, q holds the 1st node address.
                           */
        *ppPrim = (*ppPrim)->pNext; /* a new first node comes into place:
                                         * the former 2nd node now becomes the
                                         * first one.
                                         */
    } else { /* vlad: delete some node, positioned after p */
        q = p->pNext;
        if(q) p->pNext = q->pNext; /* that's how we by-pass the node after p */
    }
    free(q); /* effectively delete q.
                 * Note that it stores some address previously obtained
                 * through a call to malloc().
                 */
    return;
}

void listTraversal(pNODE pPrim, unsigned long int *pLen)
{
    pNODE tmp = pPrim; /* vlad: start with the 1st node */

    *pLen = 0;
    while( tmp != NULL )
    {
        puts(tmp->Nume);
        tmp = tmp->pNext;
        (*pLen)++;
        /* Pay attention to the operators' precedence.
         * Otherwise, you won't compute the list's length...
         */
    } /* end-while */
    return;
}

pNODE searchUltimate(pNODE pPrim)
{
    pNODE tmp = pPrim; /* vlad: start with the 1st node */
    pNODE ultimate = NULL;

    while( tmp != NULL )
    {
        if( tmp->pNext == NULL ) { ultimate = tmp; break; }
        tmp = tmp->pNext;
    } /* end-while */
    return ultimate;
}

```

```
BOOL searchValue(pNODE pPrim, char* str)
{
    pNODE tmp = pPrim; /* vlad: start with the 1st node */
    BOOL found = False;

    while( tmp != NULL )
    {
        if( strcmp(tmp->Nume, str) == 0 ) { found = True; break; }
        else tmp = tmp->pNext;
    }
    return found;
}

BOOL isEmpty(pNODE pPrim)
{
    return ( pPrim == NULL ) ? True : False;
}
```

```

// This file contains the interface implementation for Hash-tables (hash_tbls.c)
#include<stdio.h>
#include<string.h>
#include<math.h>
#include<assert.h>
#include <ctype.h>           // is...() functions

#include "hash_tbls.h"

// 'p' and 'm' are defined in the course (see lecture 6)
// Assume we want strings containing both lowercase and uppercase letters
// (which means 52 chars overall)
ULLI hash_fct(ULLI m, ULI p, char *s)
{
    ULLI sum;
    ULI i;

    sum = s[0];
    for(i = 1; i < strlen(s); i++)
    {
        if( isalpha(s[i]) )
            sum += abs(s[i] - 'a' + 1) * (ULLI)pow(p,i); // (hoping for) better indices
            //sum += abs(s[i] - 'a') * (ULLI)pow(p,i); // (hoping for) better indices
        else if( isdigit(s[i]) )
        {
            s[i] = 'A' + (s[i] - '0');      // alters the string (!)
            sum += s[i] * (ULLI)pow(p,i);
        }
    }
    return sum % m;           // here, m is a large prime number
}

ULI ins_HT(pNODE array[], char *String, ULLI m, ULI p)
{
    ULLI index; // after computation, some index is returned back by this function

    // compute the index related to the input key (the input key is a name)
    index = ( hash_fct( m, p, String ) ) % DIM;

#ifdef DEBUG
    fprintf(stdout, "String to be inserted: %s\n", String);
    fprintf(stdout, "Insert at index: %ull\n", index);
#endif

    // insert into the hash-table
    if( array[index] != NULL )
    { // search for the ultimate address...
        pNODE ultimate = NULL;
        ultimate = searchUltimate(array[index]); // I will insert at the end
        assert( ultimate != NULL );
    }

#ifdef DEBUG
    fprintf(stdout, "addr ultimate node: %p\n", ultimate);
#endif
}

```

```

    // ... and add the node:
    addNode(&array[index], ultimate, String);
} else { // the prim address of this entry is NULL => insert at the beginning
    addNode(&array[index], NULL, String);
}

(array[index]->length)++;

return index;
}

BOOL lookout_HT(pNODE array[], char *str, ULLI m, ULI p)
{
    BOOL found = False; // assume not found
    ULLI index = ( hash_fct( m, p, str ) ) % DIM;

    found = searchValue(array[index], str);

    return found;
}

void inspect_HT(pNODE array[])
{
    ULI i;

    // display the information stored at each entry
    puts("What's inside?");

    for(i=0; i<DIM ; i++)
    {
        ULI len=0;

        fprintf(stdout, "Entry: %lu\n", i);
        if( array[i] != NULL ) listTraversal(array[i], &len);
        fprintf(stdout, "Len(entry %lu): %lu", i, len);
        puts("\n");
    }
    return;
}

void listTrav_HT(pNODE pPrim)
{
    pNODE tmp = pPrim; /* vlad: start with the 1st node */
    while( tmp != NULL )
    {
        puts(tmp->Nume);
        tmp = tmp->pNext;
    }
    return;
}

```

```

// vlad: the translation unit consisting in main function (main.c)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "hash_tbls.h"

#define DEBUG
//#undef DEBUG      // get rid of DEBUG symbolic constant (if desired)

int main()
{
    pNODE array [DIM];    // vlad: this is the hash table of size DIM
                           // here, I could use dynamic allocation
    char String[64] = {0};    // this is the information I'm about to store
    ULLI m;
    ULI p;

    ULI i=0;
    ULLI index;

    // this is the initial moment (t=0) where any dynamic collection is empty
    for(int i=0; i<DIM; i++)
        array[i] = NULL; // all the lists are empty now (the beginning of time)

    m = (ULLI)1e9 + 9;    // m should be large (prime-number, since the probability
                           // of two random strings colliding is about ~ 1/m
    p = 53;   // vlad: assume the string has only lowercase and uppercase letters
               // (which means 52 chars overall => p should be a prime number
               // close to this)

    do {
        strcpy(String, "VlaD");

#ifdef DEBUG
        fprintf(stdout, "=>%lu\n", (hash_fct(p, m, "vlad") ) % DIM);
        fprintf(stdout, "=>%lu\n", (hash_fct(p, m, "Vlad") ) % DIM);
        fprintf(stdout, "=>%lu\n", (hash_fct(p, m, "dalv") & MASK) % DIM);

        fprintf(stdout, "=>%lu\n", hash_fct(p, m, "a") % DIM);
        fprintf(stdout, "=>%lu\n", hash_fct(p, m, "aa") % DIM);
        fprintf(stdout, "=>%lu\n", hash_fct(p, m, "aaa") % DIM);
#endif

        char newString[4] = {0}; // vlad: each time I'm preparing a new string,
                               // reset the working memory area (temporary!)
        sprintf(newString, "%u", i);
        newString[strlen(newString)] = '\0';

#ifdef DEBUG
        fprintf(stdout, "New string: %s\n", newString);
#endif
}

```

```

strcat(String, newString);
String[strlen(String)] = '\0';

// deal with the hash-table
index = ins_HT(array, String, m, p);
listTrav_HT(array[index]);

i++; // pass onto the next step
} while( i < DIM ); // I have at most 26 entries

// display what is stored in the table so far
inspect_HT(array);

// search for the last string added (simplified)
char Str[16] = "VlaD21"; // this string changes inside the hash-fct (!)
BOOL found = lookout_HT(array, Str, m, p);
fprintf(stdout, "Did I find (%s)? %s\n", Str, (found == True) ? "Yes":"No");

return 0;
}

```