Laboratory: STACKS
I. THEORETICAL ASPECTS

1. Introduction

A **stack** could be a simple chained list, which is managed according to the LIFO principle (Last In First Out).

According to this principle, the last node in the stack is the first one taken out. **The stack,** as a basic list, has two parts (to be indicated by two pointers): **the base of the stack** and **the top of the stack**.

On a stack we can define only three operations:

1. inserting an element in a stack (on its top) - **PUSH**;
2. removing an element from a stack (the last element that was previously added) - **POP**;
3. deleting a stack (**CLEAR**);

The first two operations are carried out at ***the top of the stack***. Thus, if an element is removed from the stack, then this element is on the top of the stack and, further, the previous inserted one reaches the top of the stack.
If an element is inserted in a stack, it will be designate to the top of the stack.

To implement a stack using a simple linked list, we must identify the base and the top of the stack, these two becoming the two extremities of the list. There are two possibilities:

a. **The node indicated by the <u>first</u> variable will be the base of the stack and the node indicated by the <u>last</u> variable will be the top of the stack;**
b. **The node indicated by the <u>first</u> variable will be the top of the stack and the node indicated by the <u>last</u> variable will be the base of the stack;**

In case **a**, the PUSH and POP functions are identified by the **add** and **erase_last_node** functions, defined in the previous laboratory. But, while the **add** function is an efficient one, the **erase_last_node** function isn't efficient.

In case **b**, the PUSH and POP functions are identified by the **inifirst** and **erase_first_node** functions (to be defined in this lesson). In this case, both functions are efficient. Thus, it is recommended to implement the stack using a simple chained list, according to case **b**.

II. ASSIGNMENT WORKFLOW

We have to solve the following problem:

In a train station, there is a train carrying goods. Its wagons are inventoried on a list, in their order. The list contains, for each wagon, the following details:

- wagon's code (9 digits);
- code of the wagon's content (9 digits);
- return address;
- recipient address.

Since in the station, the position of the wagons is reversed, then the listing of data about those wagons is required, in the new order.

In view of this, we create a stack in which we keep the data of each wagon. The data corresponding to a wagon becomes an element of the stack (a node of the simple linked list). After the data are inserted in the stack, then the elements of the list are removed and also listed during this process. This is how we can obtain the list of the wagons in reverse order.

In the main program's folder, we will define the following structure:

```
typedef struct tnod
{       long cvag;
        long cmarfa;
        int exp;
        int dest;
        struct tnod *next;
} TNOD;
```

Observation: Every function of the program will be written in a different folder with the **.cpp** extension.

1. Write a function

int pcit_int(char text[ ], int *x)

which displays a string and reads an integer from the keyboard. The function will return 0 when EOF (end of file – Ctrl+Z) is typed, and 1 otherwise.
The function has two parameters:
text – a vector of character type (the function displays this series of characters before reading the integer from the keyboard).
x – points to an integer type (the address of the memory zone in which the value of the integer is stored).

A possible solution:

```
int pcit_int(char text[ ], int *x)
/*  reads an integer and keeps it in the address of the memory zone that has the value x;
It returns:
0 - when it meets the end of the file;
1 - otherwise. */
{
 char t[255];
 for( ; ; )
 {
  printf(text);
  if(gets(t) == NULL)
       return 0;
  if(sscanf(t,"%d",x) == 1)
       return 1;
 }
}
```

2. Write a function:

int pcit_int_lim(char text[ ], int inf, int sup, int *pint)

that reads an integer that belongs to a given interval. Function has the following parameters:
text – a vector of char type (a string of characters);
inf – an integer that denotes the inferior limit of the given interval, in which we must have the number received from keyboard;
sup – an integer that denotes the superior limit of the same interval;
pint – points to integers (its value is the address of the memory zone in which the received number is stored).
The function returns:
0 – when got EOF;
1 – otherwise.

A possible solution:

```
int pcit_int_lim(char text[ ], int inf, int sup, int *pint)
/* read an integer from the interval [inf, sup] and stores it in the memory area that starts with the
parameter pint;
  - It returns:
      0 - when got EOF;
      1 - otherwise. */
{
  for( ; ; )
  {
    if(pcit_int(text, pint) == 0)
        return 0; /* s-a intalnit EOF */
    if(*pint >= inf && *pint <= sup)
        return 1;
    printf("\nThe integer typed does not belong to the range: ");
    printf("[%d,%d]\n", inf, sup);
    printf("Restart introducing\n");
  }
}
```

3. Write a function:

int incnod(TNOD *p)

that loads the current node with the data about the each wagon (wagon code, wagon content code, sender address/code, and recipient address/code).

A possible solution:

```
int incnod(TNOD *p)
/* upload a node with data about the associated wagon  */
{
 char t[255];
 char er[ ] = "EOF was typed in bad position\n";
 long cod;
 int icod;
 /* read code of the wagon */
 for( ; ; )
     {
      printf("\nCode of the wagon: ");
      if(gets(t) == 0)
        return -1;
            /* no data */
      if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
        break;
      printf("Error for wagon code\n");
     }
 p -> cvag = cod;

 /* read code of the goods */
 for( ; ; )
     {
      printf("Code of the goods: ");
      if(gets(t) == 0)
               {
                 printf(er);
                 return 0;
               }
      if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
                 break;
      printf("Error for code of the goods\n");
     }
 p -> cmarfa = cod;

 /* read sender code */
 if(pcit_int_lim("Sender code: ", 0, 9999, &icod) == 0)
  {
   printf(er);
   return 0;
  }
```

```
  p -> exp = icod;

  /* read recipient code */
  if(pcit_int_lim("Recipient code: ", 0, 9999, &icod) == 0)
   {
    printf(er);
    return 0;
   }
  p -> dest = icod;

  return 1;
} /* End incnod */
```

Observation: For this function, if we are trying to modify the input data about the sender and recipient (like the ones referring to the goods and the wagon's codes), then:
Can we simplify the function through this action? What happens with the files written in exercises 1 and 2? Comment this action.

4. Write a function:
void elibnod(TNOD *p)
that releases the node from the stack.

5. Write a PUSH function:
TNOD *inifirst()
that insert the current node before the first node of the list. The function calls the previously defined functions.

A possible solution:

```
TNOD *inifirst() /* - PUSH - insert the current node before the first node of the list */
{
 extern TNOD *first, *last;
 TNOD *p;
 int n;

 n = sizeof(TNOD);
 if(((p = (TNOD *)malloc(n)) != 0) && (incnod(p) == 1))
  {
   if(first == 0)
    {
     first = last = p;
     p -> next = 0;
    }
   else
    {
     p -> next = first;
```

```
     first = p;
    }
   return p;
  }

 if(p == 0)
  {
   printf("insufficient memory\n");
   exit(1);
  }
 elibnod(p);
 return 0;
} /* end inifirst */
```

6. Write a POP function:

void erase_first_node()

that deletes the first node from the list.

A possible solution:

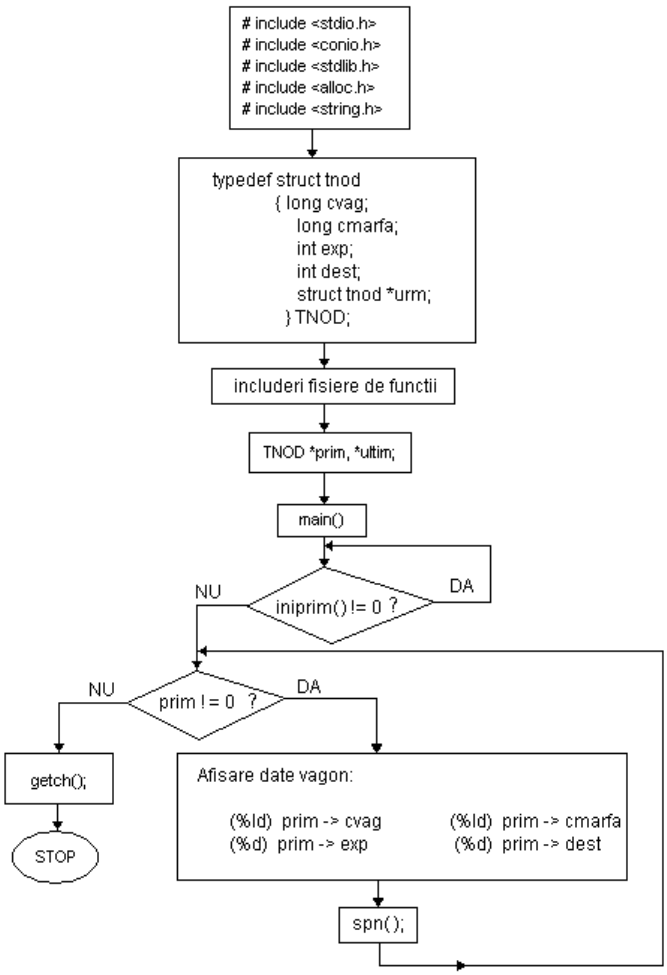
```
void erase_first_node() /* POP - delete the first node in the list */
{
 extern TNOD *first, *last;
 TNOD *p;

 if(first == 0)
       return;
 p = first;
 first = first -> next;
 elibnod(p);
 if(first == 0)
        last = 0;
} /* End erase_first_node */
```

7. Write a program that creates a stack by calling the **inifirst** function, until it returns the value 0, and then lists the wagons inventory in reverse order. It will be used the structure model defined at the beginning of this ASSIGNMENT WORKFLOW. Declare the global variables (**first** and **last)** as pointers of TNOD type.

III. ANSWERS

The flowchart of the program:



3. The modified function (from incnod_new.cpp file) is:

```
int incnod(TNOD *p)
/* upload a node with data about the associated wagon  */
{
  char t[255];
  char er[ ] = "EOF was typed in bad position\n";
  long cod;
  int icod;
  /* read code of the wagon */
  for( ; ; )
       {
```

```
      printf("\nCode of the wagon: ");
      if(gets(t) == 0)
        return -1;
            /* no data */
      if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
        break;
      printf("Error for wagon code\n");
     }
p -> cvag = cod;

/* read code of the goods */
for( ; ; )
     {
      printf("Code of the goods: ");
      if(gets(t) == 0)
                {
                  printf(er);
                  return 0;
                }
      if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 999999999)
                  break;
      printf("Error for code of the goods\n");
     }
p -> cmarfa = cod;

/* read sender code */
for( ; ; )
     {
      printf("Code of the sender: ");
      if(gets(t) == 0)
                {
                  printf(er);
                  return 0;
                }
      if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 9999)
                  break;
      printf("Error for code of the goods\n");
     }
 p -> exp = cod;

 /* read recipient code */
 for( ; ; )
     {
      printf("Code of the recipient: ");
      if(gets(t) == 0)
                {
                  printf(er);
                  return 0;
                }
      if(sscanf(t, "%ld", &cod) == 1 && cod >= 0 && cod <= 9999)
```

```
                break;
       printf("Error for code of the goods\n");
      }
   p -> dest = cod;

   return 1;
} /* End incnod_new function*/
```

In this case, there are no calls for first two functions. It has been a simplification in the implementation of the program.

4. elibnod function:

```
void elibnod(TNOD *p)
  /* Release the node pointed by p */
{
    free(p);
} /* End elibnod */
```

7. The main file of the program is:

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h> /* or alloc.h for Borland environments */
#include <stdlib.h>


typedef struct tnod
{
 long cvag;
 long cmarfa;
 int exp;
 int dest;
 struct tnod * next;
}TNOD;

#include "pcit_int.cpp" /* function from section 1 */
#include "pcit_int_lim.cpp" /* function from section 2 */
#include "incnod.cpp" /* function from section 3 */
#include "elibnod.cpp"
#include "inifirst.cpp"
#include "erase_first_node.cpp"

TNOD *first, *last;
```

```
int main() /* an inventory list of the wagons in reverse order of their reading */
{
  first = last = 0; /* The starting list is empty */

  /* inifirst function builds the stack until it returns zero */
  while(inifirst()!= 0)
        ;

  /* -  Displays the top element of the stack and then erases it from the stack;
     -  Repeat until the stack is empty */
  while(first != 0)
  {
   printf ("\nWagon code: %ld \tCode of the goods: %ld \n", first -> cvag, first -> cmarfa);
   printf ("Sender code: %d \trecipient code: %d \n", first -> exp, first -> dest);
   erase_first_node ();
  }
  getch ();
} /* End main */
```

Note: When modifying function in section 3, there shall not be included the first two files (from points 1 and 2).

When using the modified **incnode** function we have:

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h> /* or alloc.h for Borland environment */
#include <stdlib.h>


typedef struct tnod
{
 long cvag;
 long cmarfa;
 int exp;
 int dest;
 struct tnod * next;
}TNOD;

/* #include "pcit_int.cpp"
#include "pcit_int_lim.cpp"  */

#include "incnod_new.cpp" /* modified function from section 3 */
#include "elibnod.cpp"
#include "inifirst.cpp"
#include "erase_first_node.cpp"
```

```
TNOD *first, *last;

int main() /* an inventory list of the wagons in reverse order of their reading */
{
  first = last = 0; /* The starting list is empty */

  /* inifirst function builds the stack until it returns zero */
  while(inifirst()!= 0)
        ;

  /* -  Displays the top element of the stack and then erases it from the stack;
     -  Repeat until the stack is empty */
  while(first != 0)
  {
   printf ("\nWagon code: %ld \tCode of the goods: %ld \n", first -> cvag, first -> cmarfa);
   printf ("Sender code: %d \trecipient code: %d \n", first -> exp, first -> dest);
   erase_first_node ();
  }
  getch ();
} /* End main */
```

In this case, there are no calls for first two functions. It has been a simplification in the implementation of the program.