Laboratory 6: Binary trees

I. THEORETICAL ASPECTS

1. Introduction

A **tree** is somehow similar with a list, being a collection of recursive data structures that has a dynamic nature. By tree we understand a finite and non-empty group of elements called nodes:

TREE = {A1, A2, A3, ..., An}, where **n> 0**,

which has the following properties:

- there is only one node, which is called the root of the tree.
- the rest of the nodes can be grouped in **subsets** of the initial tree, which also form trees. Those trees are called **subtrees** of the root.

In a tree there are nodes which have no descendants. Such a node is called terminal node or leaf. Some useful terminologies:

- **Root** – The top node in a tree;
- **Parent** – The converse notion of child;
- **Siblings** – Nodes with the same parent;
- **Descendant** – a node reachable by repeated proceeding from parent to child;
- **Ancestor** – a node reachable by repeated proceeding from child to parent;
- **Leaf** – a node with no children.

2. Binary tree

A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. There are maximum two disjoint groups for every parental node (each one being a binary tree). Sometimes, one of the groups is called the left subtree of the root, and the other one the right subtree. The binary tree is ordered, because in each node, the left subtree is considered to precede the right subtree. In other words, we can say that the left descendant is older that the right one.

Sometimes, a node of a binary tree can have only one descendant. This can be the left subtree or the right subtree. (**NOTE**: The two possibilities are considered distinct)

A binary tree it cannot be defined as a particular case of an ordered tree. Usually, a classic tree is never empty, while a binary tree can be empty, sometimes. Any ordered tree can be always represented through a binary tree. With this in mind, we will continue only with binary trees, since this covers all the possibilities.

The node of a binary tree can be represented as another structural data type, called NOD, which is defined as follows:

```
typedef struct nod
{       <statements>
        struct node * left;
        struct node * right;
} NOD;
```

where:

left - is the pointer to the left son of the current node;

right - is the pointer to the right son of the same node.

In applications with binary trees we can define several operations such as:

- Inserting a leaf node in a binary tree;
- Access to a node of a tree;
- Traversal the tree;
- Delete a tree.

The operations of insertion and access to a node are based on a **criteria** that defines the place in the tree where the node in question can be inserted or found (according with the current operation which is involved). This **criterion** is dependent on the specific problem where the binary tree concept is applied.

It will be defined in the next section (II. ASSIGNMENT WORKFLOW) a function which will be called the criterion function. This function has two parameters, which are pointers of NOD type. Considering p1 as the first parameter of the criterion function and p2 the second one, then the criterion function will return:

-1 - if p2 indicates to a data of NOD type which can be inserted in the left subtree of the node pointed by p1;
1 - if p2 indicates to a data of NOD type which can be inserted in the right subtree of the node pointed by p1;
0 - if p2 is equivqlent with p1.

When we are building a tree, it is established a criterion to find the position in which will be inserted the new current node in the tree – i.e. for the corresponding node of the last acquired value (or set of values).
For example, it is considered:
a. **p1** - is a pointer to a node from the tree to which the inserting is to be linked (p1 indicates initially to the root of the tree)
b. **p2** – is a pointer to the current node (the node that will be inserted)
c. if **p2->val < p1->val**, then it tries to insert the current node into the left subtree of the node indicated by p1
d. if **p2->val > p1->val**, then it tries to insert the current node into the right subtree of the node indicated by p1
e. if **p2->val = p1->val**, then the current node will not be inserted in the tree, because it already exists a corresponding node for the current value.

These tasks will be fulfilled by the **criterion** function (see section II Assignment Workflow).
The current node is no longer inserted in the tree in the **e** case (when the **criterion** function returns zero). In this case, the nodes pointed by p1 and p2 we consider to be **equivalent**.
Usually, when we have two equivalent nodes, **p1** is incremented (or processed in a specific way) and **p2** is eliminated. To achieve such processing is necessary to call a function that takes as parameters the pointers **p1** and **p2**, and returns a NOD type pointer (usually returns the value of **p1** after deleting the **p2**). We call this function: **equivalence**. It is dependent on the specific issue to be solved by the program.

In addition to the functions listed previously, we also use other specific functions for operations on binary trees. Typical examples of functions are **elibnod** and **incnod**.
Some functions use a global variable that is a pointer to the root of the tree. We denote **proot** a global variable to the root of the binary tree. It is defined as:

NOD *proot;

NOTE: Next we use some functions based on the global variable **proot**.

Inserting a leaf node in a binary tree

The function **insnod** inserts a node in the tree, according to the following steps:

1. It is allocated a memory area for the node to be inserted in the tree. Consider **p** being the pointer for this memory.

2. By calling the **incnod** function, we have to fill the node with data. If **incnod** returns 1, then jump to step 3. Otherwise the function returns the value zero.

3. Assignments are made:
p->left = p->right = 0
since the new node is a leaf one.

4. **q = proot**

5. Find the position in the tree where the insertion will be made (find the possible parent for the node which will be inserted):
i = criterion(q, p)

6. If **i<0**, then jump to step 7; otherwise jump to step 8.

7. Try to insert the current node to the left subtree of the root **q**.
- If **q -> left** is zero, then the current node becomes the left leaf of q (**q->st = p**). Afterwards the function returns the value of **p**.
- Otherwise **q = q->st** , and jump to step 5.

8. If **i>0**, then jump to step 9; otherwise jump to step 10.

9. Try to insert the current node to the right subtree of the root **q**.
- If **q -> right** is zero, then the current node becomes the right leaf of q (**q->right = p**). Afterwards the function returns the value of **p**.
- Otherwise **q = q->right** , and jump to step 5.

10. The current node cannot be inserted into the binary tree. In this situation we have to call the **equivalence** function.

Tree traversal
Tree traversal is a form of graph traversal and refers to the process of visiting (examining or updating) each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Pre-order
- Display the data part of root element (or current element)
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.

In-order (symmetric)
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of root element (or current element).
- Traverse the right subtree by recursively calling the in-order function.

Post-order
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of root element (or current element).

The access to a node allows the processing of the information contained in the respective node. For this you can call a function that is dependent of the specific problem that implies traversal of the tree (see the process function from the next section).

II. ASSIGNMENT WORKFLOW

We have to write a program that reads the words from a text and displays the number of occurrences of each word from this text. The word is defined as a succession of upper and lower cases. The text ends when CTRL+Z are typed.

This problem was previously solved with a simple linked list. This time we will use a binary tree.

NOTE: The functions or programs from each point of this section will be written in separate files with the extension **.cpp**.

1. Write a function that reads one word and keeps it in the heap memory (a word means a succession of uppercase and lowercase letters). The function returns the starting address of the memory zone in which the word is maintained, or zero in the case of EOF (Ctrl + Z).

A possible solution:

```
char *citcuv()
/* - Reads a word and keeps it in the heap memory;
- Returns the pointer to that word or zero in the case of EOF   */
{
 int c, i;
 char t[255];
 char * p;
/* skip over characters that are not letters */
 while ((c = getchar ()) <'A' || (c> 'Z' && c <'a') || c> 'z')
 if (c == EOF)
 return 0; /* EOF case */
/* read a word and keep it in the vector t */
 i = 0;
 do
 {
 t[i ++] = c;
 } while (((c = getchar()) >= 'A' && c <= 'Z' || c >= 'a') && c <= 'z');
```

```
 if (c == EOF)
         return 0;
 t[i++] = '\0';
/* the word is saved in the heap memory */

 if ((p = (char *) malloc(i)) == 0)
   {
    printf("Insufficient memory \n");
    exit (1);
   }
 strcpy (p, t);
 return p;
}
```

Note: **strcpy** function was studied during previous semester

2. It is consider the following user type:

```
typedef struct nod
                {
                 char *word;
                 int frequency;
                 struct nod *left;
                 struct nod *right;
                } NOD;
```

which will be used in all the next assignments.

It is required to write a sort of **incnod** function, which loads the current data in a TNOD type node. This function calls the **citcuv** function and assigns the returned address of it to the word pointer from the current node. Also, it is assigned the value 1 to the frequency variable. The **incnod** function returns the value -1 if **citcuv** returns 0, otherwise returns 1.
A possible solution:

```
int incnod(NOD *p)
/* tries to load the current data in the node p and shows if it was done successfully */
{
  if ((p->word = citcuv()) == 0) return -1;
  p -> frequency = 1;
  return 1;
}
```

3. Write a function that releases the zones from the heap memory allocated by the node that was defined in the previous exercise.

A possible solution:

```
void elibnod(NOD *p)
/* Release the heap memory areas allocated by a pointer type node p */
{
  free(p->word);
  free(p);
}
```

4. Write a function called equivalence:

NOD *equivalence(NOD *q, NOD *p)

which fulfills the following tasks:

- frees the memory area pointed by **p**
- increments the frequency from **q**
- returns **q**

5. Write the function:

void process(NOD * p)

which displays the data from a node type. Assuming that the data will be displayed continuously, you have to introduce a waiting from time to time at each 23 rows.

A possible solution:

```
void process(NOD *p) /*  display p -> word
                              and       p -> frequency */
{
 static int n = 0;

 printf("\nThe word: %s has the frequency: %d", p->word, p->frequency);
 if((n+1)%23 == 0)
          {
           printf("\nPress a key to continue");
           getch();
          }
 n++;
}
```

6. Write the function:

int criterion(NOD *p1, NOD *p2)

This function has two parameters, which are pointers of NOD type. Considering p1 as the first parameter of the criterion function and p2 the second one, then the **criterion** function will return:

-1 - if p2 indicates to a data of NOD type which can be inserted in the left subtree of the node pointed by p1;

1 - if p2 indicates to a data of NOD type which can be inserted in the right subtree of the node pointed by p1;

0 - if p2 is equivalent with p1.

A possible solution:

```
int criterion(NOD *p1, NOD *p2)
        /* it returns :
                -1      - if p2->word < p1->word;
                 1      - if p2->word > p1->word;
                 0      - otherwise.  */
{
 int i;

 if((i=stricmp(p2 -> word, p1 -> word)) < 0)
        return -1;
 else
        if(i>0)
                return 1;
        else
                return 0;
}
```

7. Write a function:

NOD * insnod ()

which according to the steps described in the first section inserts a node (of NOD type) in a binary tree.

A possible solution:

```
NOD *insnod()
/* - inserts a node in a binary tree pointed by proot;
   - returns a pointer to the inserted node,
     or a pointer returned by the function equivalence;
   - returns zero if there are no data to load, or when an error occurred.  */
{
 extern NOD *proot;
 int i;
 int n;
 NOD *p, *q;
 n = sizeof(NOD);
```

```
if(((p = (NOD *)malloc(n)) != 0) && (incnod(p) == 1)) /* 1 */
        {
         p -> left = p -> right = 0;

         if(proot == 0)  /* empty the tree */
                  {
                   proot = p;
                   return p;
                  }

         q = proot;

         for( ; ; )
                  {
                   if((i = criterion(q,p)) < 0)
                            if(q -> left == 0)
                                     {
                                      q -> left = p;
                                      return p;
                                     }
                            else
                                     {
                                      q = q -> left;
                                      continue;
                                     }

                   if(i > 0)
                            if(q -> right == 0)
                                     {
                                      q -> right = p;
                                      return p;
                                     }
                            else
                                     {
                                      q = q -> right;
                                      continue;
                                     }

                   return equivalence(q,p);
                  }  /* ends for */
        }  /* ends if 1 */

if(p == 0)
        {
         printf("\n Insufficient memory\n");
         getch();
         exit(1);
        }
elibnod(p);
return 0;
}
```

8. Write the function:

void inord (NOD * p)

which traverses a binary tree in inorder.

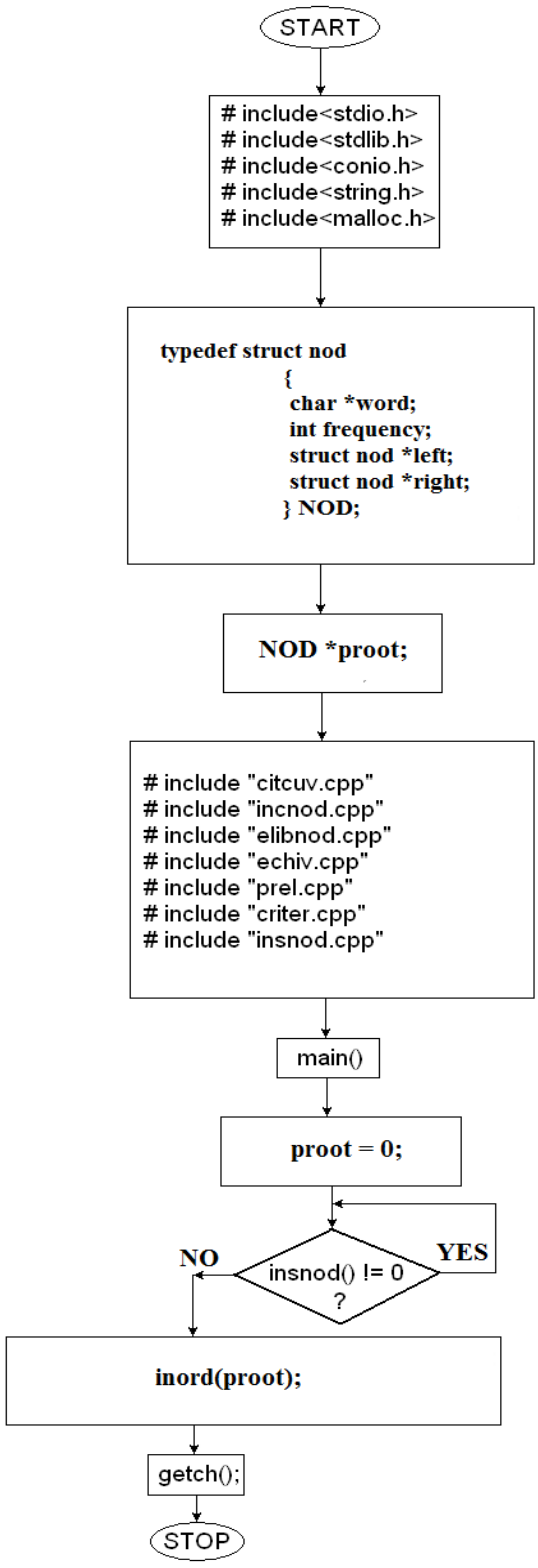
A possible solution:

```
void inord(NOD *p)  /* traverse a binary tree in inorder */
{
 if(p != 0)
          {
           inord(p -> left);
           process(p);
           inord(p -> right);
          }
}
```

9. Write a program that reads the words from a text and displays the number of occurrences of each word from this text. The word is defined as a succession of upper and lower cases. The text ends when CTRL+Z are typed. It will be used the model of the NOD structure (see point 2) and it is necessary to include all functions defined at the previous points. The global variable to the root of the tree is **proot**. The program will build a binary tree, which will be finally traversed in inorder (displaying each word together with its frequency).

III.ANSWERS

The flow chart of the program is:



4. Equivalence function:

```
NOD *equivalence(NOD *q, NOD *p)
/* - releases the memory indicated by the pointer p;
   - adds a unit to  q -> frequency;
   - returns the value of q.  */
{
 elibnod(p);
 q -> frequency++;
 return q;
}
```

9. The main file:

```
# include<stdio.h>
# include<stdlib.h>
# include<malloc.h>
# include<conio.h>
# include<string.h>

typedef struct nod
                    {
                     char *word;
                     int frequency;
                     struct nod *left;
                     struct nod *right;
                    } NOD;

# include "citcuv.cpp"
# include "incnod.cpp"
# include "elibnod.cpp"
# include "equivalence.cpp"
# include "process.cpp"
# include "criterion.cpp"
# include "insnod.cpp"
# include "inord.cpp"

NOD *proot;

int main()  /* Reads the words in a text and displays them in alphabetical order
along with their frequency of occurrence in the text */
{
 proot = 0;
/* builds the binary tree */
 while(insnod())
          ;

 inord(proot); /* traverse the binary tree in inorder */
 getch();
}
```