University of Nantes
Master 2 in Bioinformatics

Pedagogical support

Algorithms for trees

Christine Sinoquet

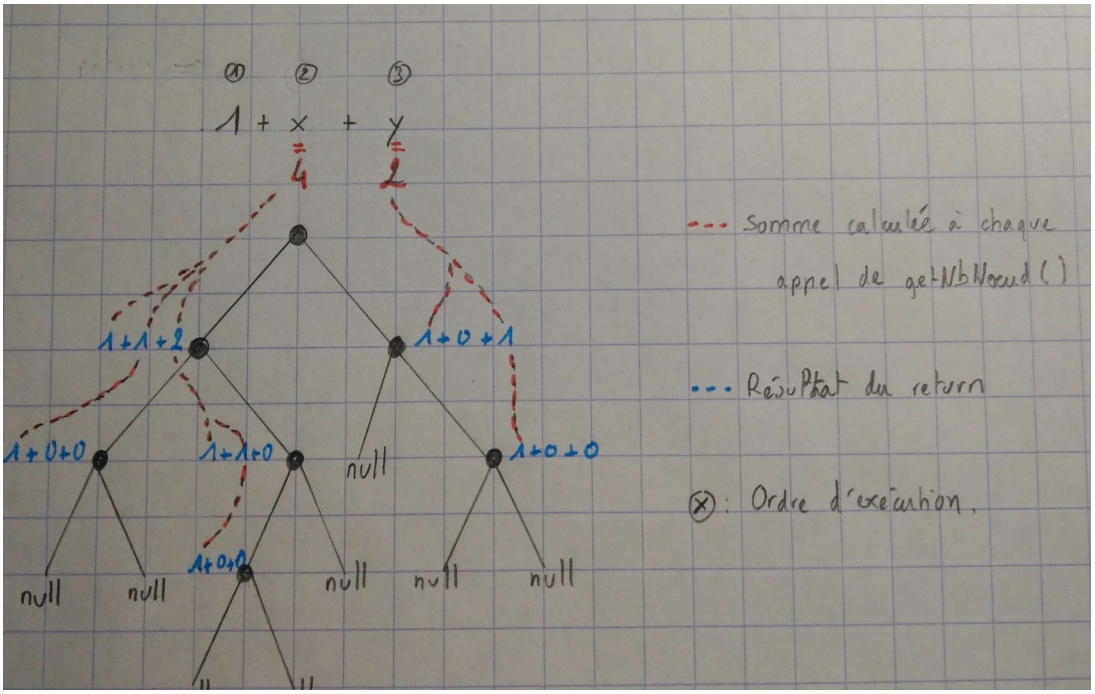
TABLE OF CONTENTS

```
typedef struct {
        int val;
        struct TNode* lc; // left child
        struct TNode* rc; // right child
} TNode;
typedef  TNode* TBinTree;
```

Computation of the number of nodes in a binary tree
Recursive version in C, version 1

```
/*********************************************************/

int getNbNodes (TBinTree n){

    if  ( n == NULL) {return 0;}
    return (1 + getNbNodes(lc(n)) + getNbNodes(rc(n));
}

/*********************************************************/

void main(){

int          res;
TBinTree r;
r = create_binary_tree(...) // creation of a binary tree (not detailed here)

res = getNbNodes(r);

/*********************************************************/
```

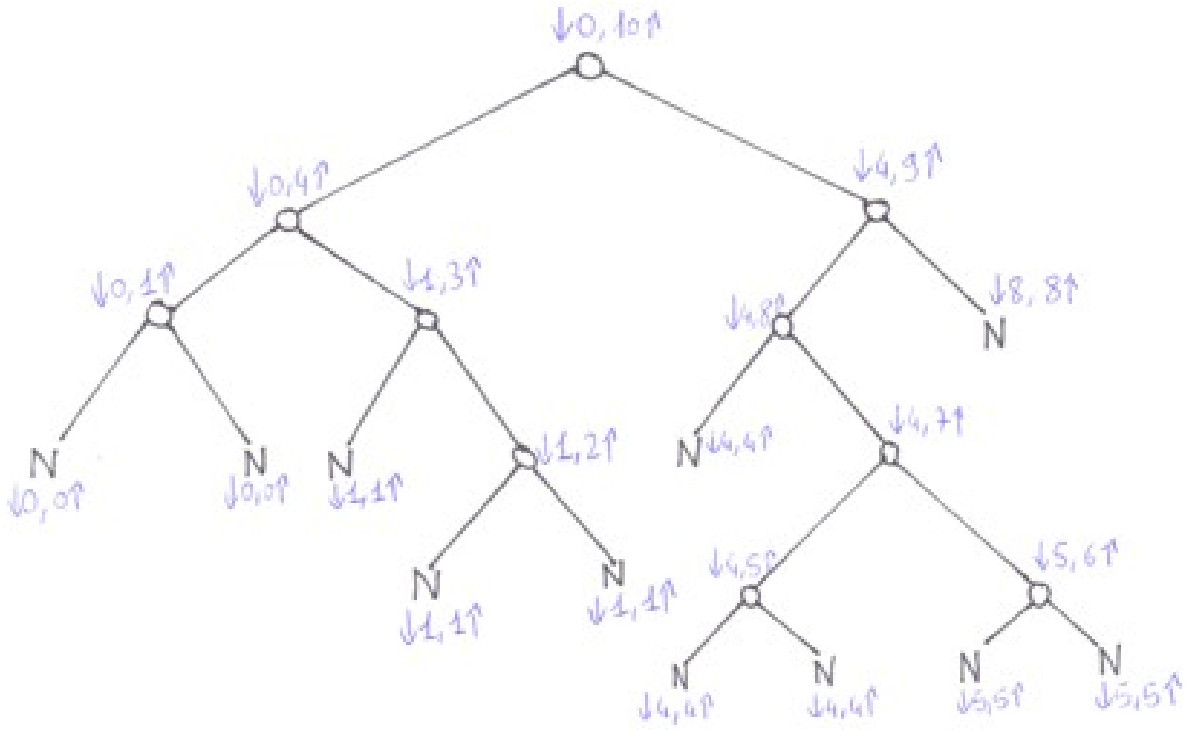
Execution trace for a toy example:



Computation of the number of nodes in a binary tree
Recursive version in C, version 2

```
/*********************************************************/

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

typedef struct TNode {
 int val;
 struct TNode* fg;
 struct TNode* fd;
} Tnode;

/*********************************************************/

void nb_nodes_bis(TNode* n, int* adr_cnt) {

 if ( n != NULL ) {
  nb_nodes_bis(n->lc, adr_cnt);
  nb_nodes_bis(n->rc, adr_cnt);
  *adr_cnt = *adr_cnt + 1;
 }
} # end of  nb_nodes_bis

/*********************************************************/

void main() {

 int cnt = 0;
 TNode*   root        = (Tnode*) malloc(sizeof(TNode));
 TNode*   l_child     = (Tnode*) malloc(sizeof(TNode));
 TNode*   r_cchild    = (Tnode*) malloc(sizeof(TNode));

 root→ val = 10;
 root→lc    = l_child;
 root→rd   = r_child;

  l_child → val = 21;
  l_child → lc   = NULL;
  l_child → rc   = NULL;

  r_child → val = 22;
  r_child → lc   = NULL;
  r_child → rc   =NULL;

 nb_nodes_bis(root, &cnt);

 printf("The number of nodes is %d.\n", cnt);
} # end main

/*********************************************************/
```

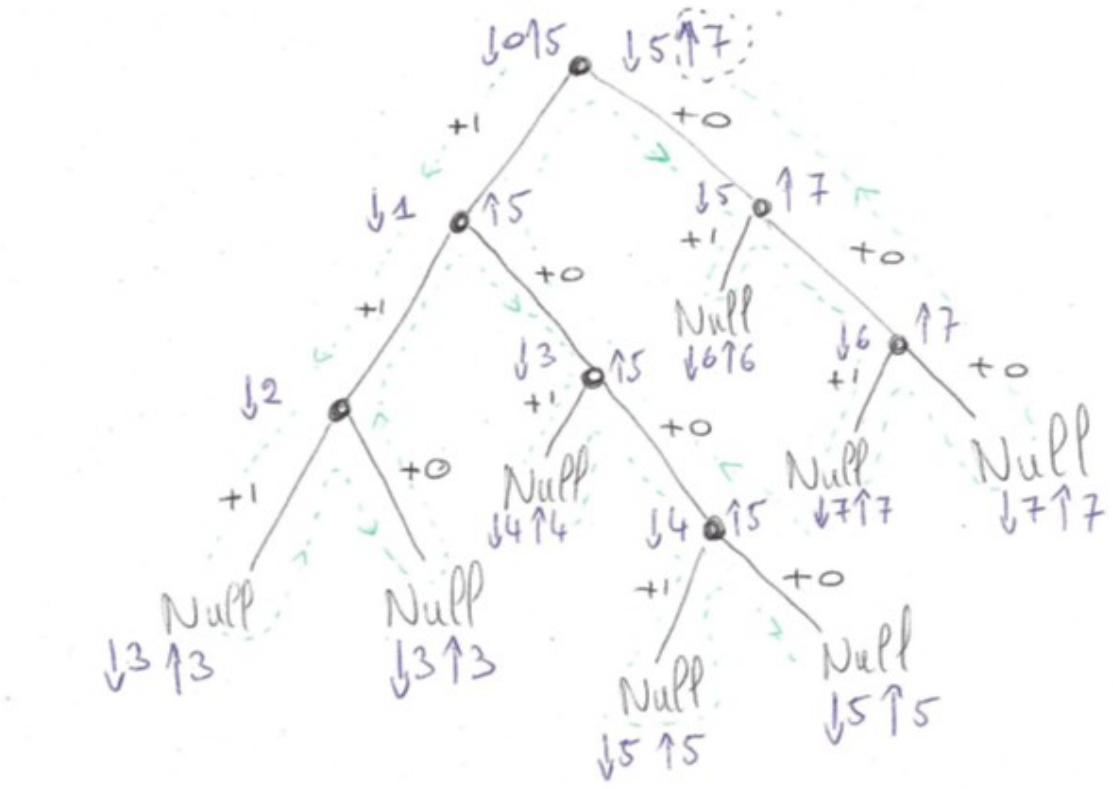
Execution trace for a toy example:



Computation of the number of nodes in a binary tree
Recursive version in C, version 3 with accumulator

```
/*********************************************************/

void nbNodesTer(TBinTree n, int cnt, int* adr_res){
 int res_left;

   if ( n == NULL) {
    *adr_res = cnt;
   } else {
    nbNodesTer(left(n), 1 + cnt, &res_left); // second parameter is passed by value
                                             // third parameter is passed by address
    nbNodesTer(right(n), *res_left, adr_ res);
  }
} // end nbNodesTer


/*********************************************************/

void main(){

TBinTree root = create_binary_tree(...) // creation of a binary tree (not detailed here)
 int          cnt   = 0; // initialization of counter
 int res;

nbNodesTer(root, cnt, &res) ;
} // end main
/*********************************************************/
```

Execution trace for a toy example:



Retrieval of an information in a binary tree
Recursive version in C

```
/*********************************************************/

void search_val (TBinTree n, int val, bool* adr_found){

    if (n != NULL){
        if (n→val == val) {
                *adr_found = true;
        } else {
                search_val(left(n), val, adr_found);
                if !(*adr_found) {
                        search_val(right(n), val, adr_found);
                }
        }
    }
} // end  search_val

/*********************************************************/

void main(){
 bool        found = false;
 int         val     = 42; // searched value
TbinTree   root = create_binary_tree(...) // creation of a binary tree (not detailed here)

search_val(root, val, &found);
} // end main

/*********************************************************/
```

Retrieval of an information in a binary tree, with mention of the physical address of this information if existing
Recursive version in C

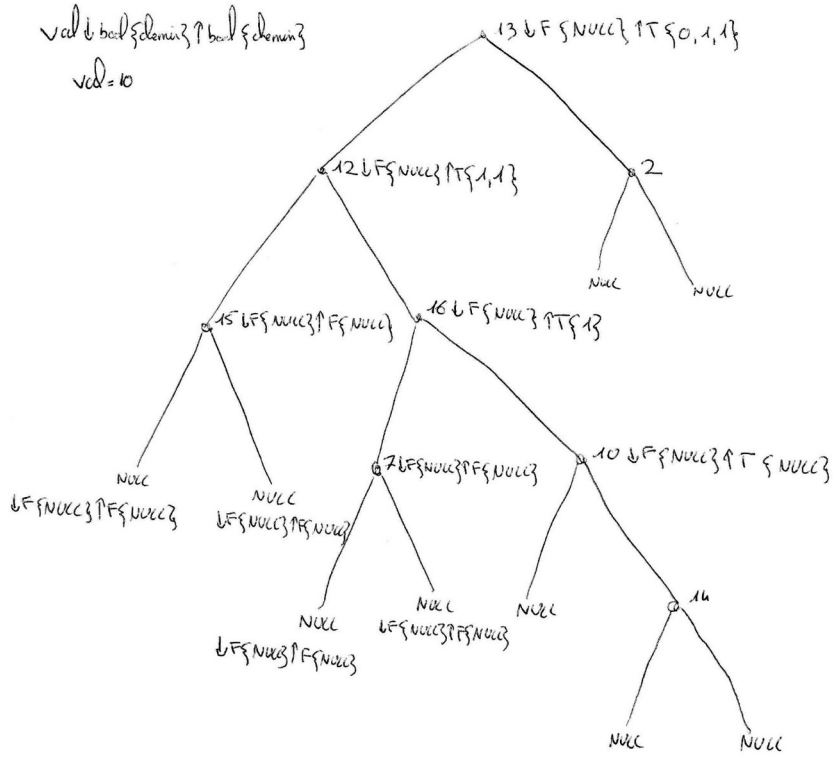
```
/*********************************************************/

void search_val (TBinTree n, int val, TBinTree* adr_ptr_val, bool* adr_found){
// precondition:
// *adr_found == false;

        if(n != NULL){
                if(n→val == val){
                        *adr_found = true; *adr_ptr_val = n;
                }else{
                        search_val(left(n), val, adr_ptr_val, adr_found);
                       if (! (*adr_found)){
                        search_val(right(n), val, adr_ptr_val, adr_found);
                       }
                }
        }
} // end search_val

/*********************************************************/

void main(){
 int          val = 450;
 TBinTree root;
 TbinTree  ptr_val;
  bool       found = false;

  root = create_binary_tree(...) // creation of a binary tree (not detailed here)

  found = search_val(root, val, &ptr_val, &found);
} // end main



/*********************************************************/
```

Retrieval of an information in a binary tree, with computation of the path towards this information if existing

Recursive version in C, version 1

```
/*********************************************************/

bool search_val_binary_tree(TBinTree n, int val, Tpointer*  adr_head_list)
{
 bool found = false;

 if (n == NULL) {  return false; }
 if (n→val == val) { return true; }
 // n != NULL and n→val != val
 found = search_val_binary_tree(n → lc, val, adr_head_list); //exploring subtree rooted in left child
 if (found){ addFront(adr_head_list, 0);} // 0 is added to the path under construction,
                                          // which means that the information val was found when
                                          //  moving to left child
 else
 {
   found = search_val_binary_tree(n → rc, val, adr_head_list); //exploring subtree rooted in right child
   if (found) {addFront(adr_head_list, 1);} // 1 is added to the path under construction,
                                            // which means that the information val was found when
                                            //  moving to right child
 }
 return found;
} // end  search_val_binary_tree

/*********************************************************/

void main(){

TBinTree root = create_binary_tree(...) // creation of a binary tree (not detailed here)
Tpointer  list   = NULL;
val=10;
bool found = search_val_binary_tree(root, val, &list);
if ((found) && (list==NULL)) { addFront(&liste, 2);} // to handle the case when the information
                                                     //  is contained in the root
} // end main

/*********************************************************/
```

Execution trace for a toy example:


13 ↓F {NULL} ↑T {0,1,1}
12 ↓F {NULL} ↑T {1,1}
2
NULL
NULL
15 ↓F {NULL} ↑F {NULL}
16 ↓F {NULL} ↑T {1}
NULL
↓F {NULL} ↑F {NULL}
NULL
↓F {NULL} ↑F {NULL}
7 ↓F {NULL} ↑F {NULL}
10 ↓F {NULL} ↑T {NULL}
NULL
↓F {NULL} ↑F {NULL}
NULL
↓F {NULL} ↑F {NULL}
NULL
14
NULL
NULL

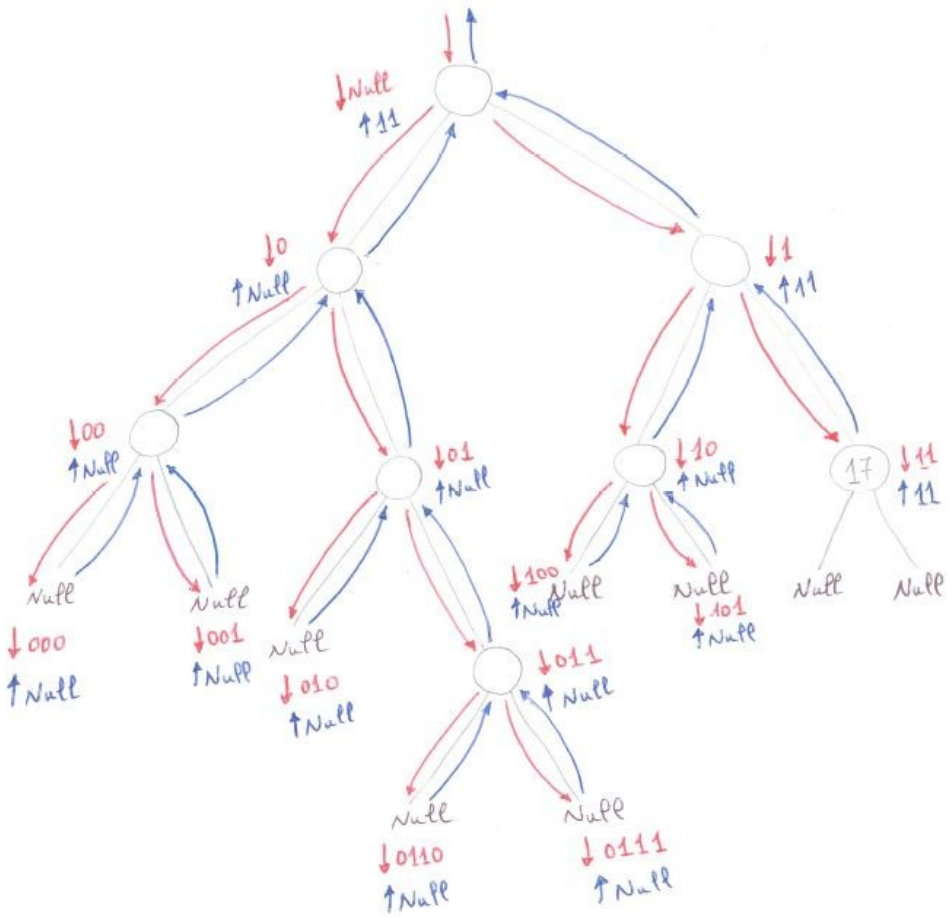
/**/

Retrieval of an information in a binary tree, with computation of the path towards this information if existing

Recursive version in C, version 2 with accumulator

```
TPointer search_val_binary_tree_bis (TBinTree n, int val, TPointer acc){ // acc = accumulator
  if (n==NULL){ return NULL;}
  if (n → info == val){ return acc; } // The path towards the value is stored in the list acc.
  res = search_val_binary_tree(n → lc, val, novelListQ(acc, 0));
                                                                    // (*) The function NovelListQ returns
                                                                    // a novel list obtained by the
                                                                    //  concatenation of acc and of a list
                                                                    // composed of 0.
                                                                   //  The list acc is left unchanged.
                                                                  // 0 means that the path towards
                                                                 // information val traversed n's left child.
  if (res != NULL){ return res;}
  res = search_val_binary_tree(n → rc, val, novelListQ(acc,1)); // see (*) above
                                                                     // 1 means that the path towards
                                                                    // information val traversed n's right
                                                                    // child.
  return res;
} // end  search_val_binary_tree_bis

/*********************************************************/

void main(){

TBinTree root = create_binary_tree(...) // creation of a binary tree (not detailed here)
Tpointer  acc   = NULL; // initialization of accumulator
Tpointer  res;
int          val = 17;
if ((root != NULL) && (root →  val == val)){  // to handle the case when the information
                                                 //  is contained in the root
    res = NULL; addFront (&res, 2);
 } else {
    res = search_val_binary_tree_bis (root, val, acc);
     print_list((res));
 }

} // end main

/*********************************************************/
```

Execution trace for a toy example:



Retrieval of an information in a search binary tree

Recursive version in C, version 1

```
/*********************************************************/
bool getValBST(TBST n, int val){
// precondition:
// For each n1 in subtree rooted in left child of n: val(n1) ≤ val(n)
// For each n2 in subtree rooted in right child of n: val(n) < val(n2)

    bool found;
    found = false;

    if (n == NULL) {
        return false;
    }

    if (n→val == val){
        return true;
    } else {
        if (n→val  < val){
                found = getValBST(right(n), val);
        } else {
                found = getValBST(left(n), val);
        }
    }
} // end getValBST

/*********************************************************/


void main(){
 int      val     = 42; // searched value
 TBST root    = create_binary_tree(...) // creation of a binary tree (not detailed here)
 bool   value_is_found;
 value_is_found = getValBST(root, val);
} // end main

/*********************************************************/
```

Retrieval of an information in a search binary tree, with computation of the path towards this information if existing

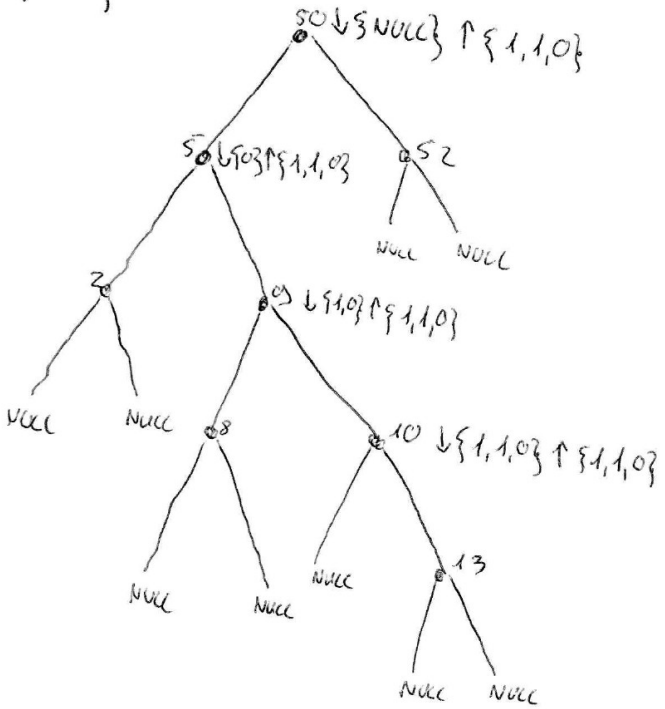
Recursive version in C, version 2 with accumulator

```
/*******************************************************/
TPointer search_val_in_binary_search_tree(TBST n, int val, TPointer acc)
// precondition:
// For each n1 in subtree rooted in left child of n: val(n1) ≤ val(n)
// For each n2 in subtree rooted in right child of n: val(n) < val(n2)
{
        if (n=NULL){return NULL;}
        if (n→val == val) {return acc;}
        if (n→val >=val)
        {
           return(search_val_in_binary_search_tree(n→lc, val, novel_list_head(acc, 0));
                                                    // (*) The function novel_list_head returns
                                                    // a novel list obtained by the
                                                    //  concatenation of a list composed
                                                    // of 0 and of list acc.
                                                    //  The list acc is left unchanged.
                                                    // 0 means that the path towards
                                                    // information val traversed n's left child.
        }
        return (search_val_in_binary_search_tree(n→rc, val, novel_list_head(acc, 1));
                                                    // see (*) above
                                                    // 1 means that the path towards
                                                    // information val traversed n's right
                                                    //  child.
} // end  search_val_in_binary_search_tree

/*******************************************************/
void main(){

 TBST    root  = create_binary_tree(...) // creation of a binary tree (not detailed here)
 int        val    = 10; // searched value
TPointer acc   = NULL; // initialization of accumulator
TPointer path_towards_val = search_val_in_binary_search_tree(root, val, acc);

if (path_towards_val!= NULL){
 print_list(reverse(path_towards_val)) ;
}

} // en main

/*******************************************************/
```

Execution trace for a toy example:

val : 10

val : {chemin} ↑ {chemin}

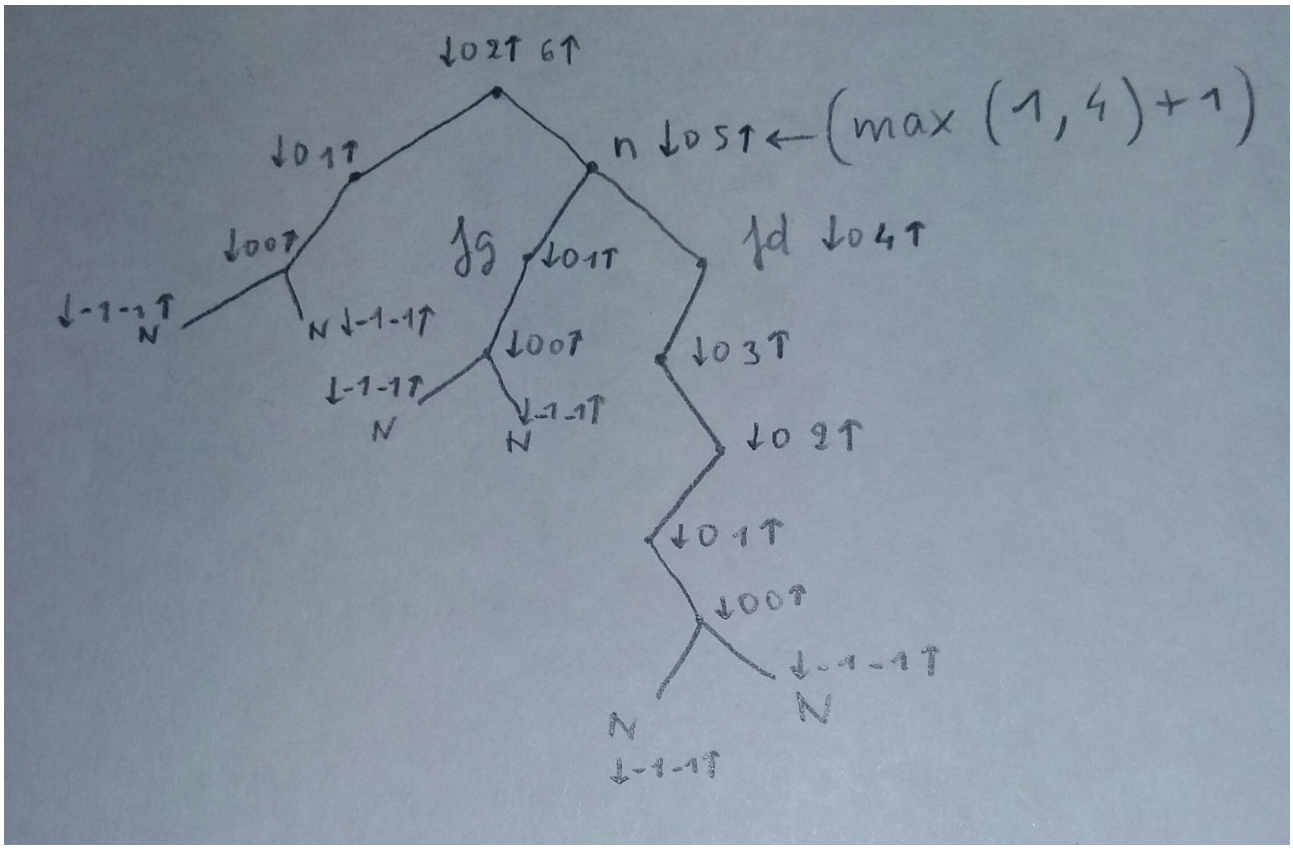


Remarque : chemin en arche inverse dans la liste

Computation of the height of a binary tree
Recursive version in C, version 1

```
/*********************************************************/

int height(TBinTree n){
     if (n == NULL){
           return(-1);
      }
     return(1 + max(height(n→lc), height (n→rc)));
} // end height

/*********************************************************/

void main(){

 TbinTree root1  = create_binary_tree(...) // creation of a binary tree (not detailed here)
  TbinTreeroot2  = NULL ;

  int   tree_height_1 = height(root1);
  int   tree_height_2 = height(root2); // An empty tree has a height equal to -1.

} // end main

/*********************************************************/
```

Execution trace for a toy example:



Computation of the height of a binary tree
Recursive version in C, version 2 with accumulator

```
/*********************************************************/

void height_bis (TBinTree n, int cnt, int* adr_height){
 if (n == NULL){
  if (cnt > *adr_height){
   *adr_height = cnt; // The height computed so far (*adr_height) is replaced with the length of the
                      //  current tree branch (cnt).
  }
  return;
 }
 // postcondition:
 //   n != NULL
  height_bis (n→lc, cnt+1, adr_height);
  height_bis (n→rc, cnt+1, adr_height);
} // end  height_bis

/*********************************************************/
void main(){
 int height;
 int cnt;
 TBinTree root = create_binary_tree(...) // creation of a binary tree (not detailed here)

  height = -1;
  cnt      = -1;
  height_bis (root, cnt, &height);

} // end main

/*********************************************************/
```