SDD : TP 3

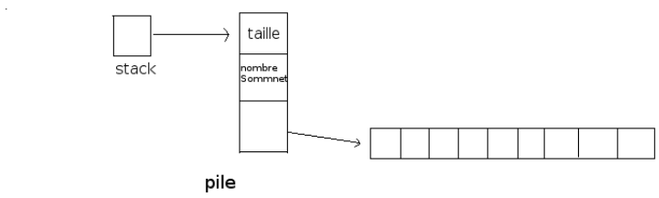
Mathieu Boutin - Jérémy Morceaux

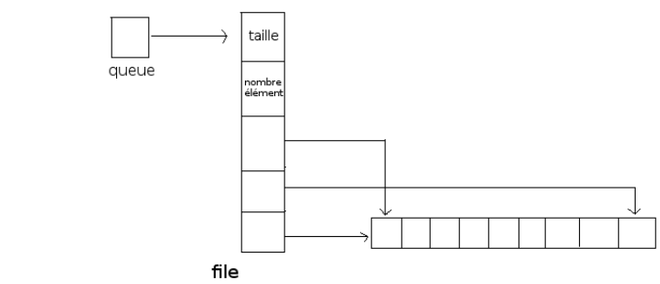
June 11, 2018

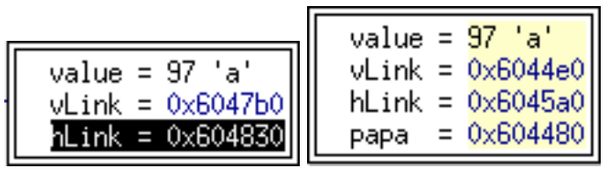
1 Présentation générale

- Ce TP a pour but de travailler sur la représentation des arbres en mémoire. Tout d'abord, il fallait créer un arbre en utilisant la représentation lien vertical/horizontal. Puis de vérifier le résultat avec le débuggeur ddd. Ensuite, il fallait afficher cette arbre avec la représentation postfixée. Par la suite, il fallait créer une méthode d'insertion dans l'arbre. Il fallait également créer un autre type d'arbre où chaque noeud possède un pointeur vers le noeud père, à partir de l'arbre de base. Finalement, il fallait afficher la représentation postfixée de ce nouvel arbre.

- Schéma de base :

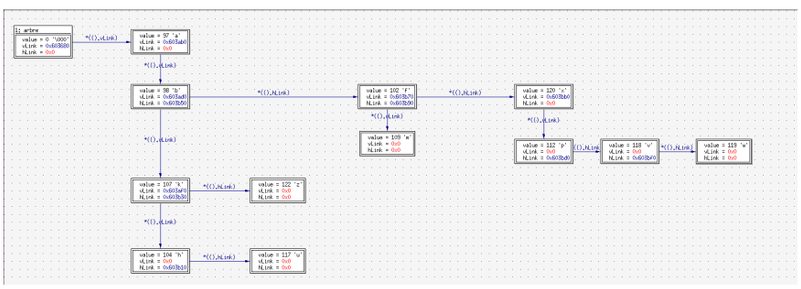




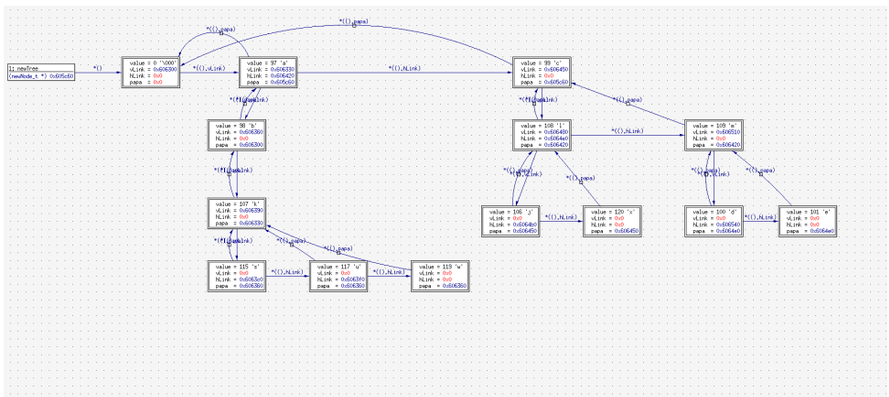


noeud de l'arbre de base / noeud de l'arbre générée à partir de l'arbre de base

Vérification avec DDD :



Arbre de base obtenu grâce à DDD



Arbre avec lien vers le père obtenu grâce à DDD

2 Détail de chaque fonction

2.1 main

Principe : main

On crée un arbre à partir de la chaîne de caractère treeString.
Si la création a réussi:
    Si l'arbre est non-vide:
        On affiche la représentation postfixée.
        On recherche le noeud sur lequel on veut insérer le nouveau noeud.
        Si on a trouvé le noeud.
            On insère celui-ci dans l'arbre
            Si l'insertion a réussi:
                On affiche un message de compliment.
            Sinon
                On affiche un message d'erreur.
        On affiche la représentation postfixée du nouvel arbre.
        On crée un nouvel arbre à partir de l'arbre de base.
        Si la création a réussi:
            On affiche la représentation postfixée de ce nouvel arbre.
        Sinon
            On affiche un message d'erreur.
        On libére l'arbre de base.
        On libère l'arbre généré à partir de l'arbre de base.
    Sinon
        On affiche un message d'erreur.
Sinon
    on affiche un message d'erreur
On libère la tête de l'arbre de base
FIN

Lexique :

- Variables locales:
  - arbre: c'est le pointeur vers l'arbre de base.
  - errorCode: indique si une fonction s'est bien déroulée.
  - treeString: chaine de caractère permettant la création de l'arbre de base.
  - pere: pointeur vers le noeud du père du noeud à insérer.
  - newtree: pointeur vers l'arbre généré à partir de l'arbre de base.
  - p: caractère du père pour le nouveau noeud à insérer
  - i: caractère du noeud à insérer

Programme Commenté :

```c
/* ############################################################ */
/* file's name : main.c                                         */
/*                                                              */
/* Author : Mathieu Boutin & Jeremy Morceaux                    */
/* Date : March 2018                                            */
/*                                                              */
/* This file contains the main program of the TP3               */
/* ############################################################ */


#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

#define ROUGE  "\x1B[31m"
#define VERT  "\x1B[32m"
#define BLANC  "\x1B[37m"

#include "ZZ_tree.h"

int main()
{
   noeud_t* arbre;
   int errorCode;
   char *treeString = "(a(b(k(h,u)z)f(m)x(p,v,w)))";

   printf("%sI/ Creation de l'arbre !\n%s",VERT,BLANC);
   arbre = createTree(treeString,&errorCode);

   if(errorCode == 0)
   {
      if(arbre->vLink != NULL)
      {
         noeud_t*   pere;
         newNode_t* newTree;
         char       p,i;

         printf("Creation réussie !\n");
         printf("%sII/ Premiere representation Postfixee avant insertion: \n%s",VERT,
             BLANC);

         /* display the postfix notation of this tree , startint at vLink because we don'
             t want to display the head of the tree */
         repPostFixe(arbre->vLink,&errorCode);

         /* unable to display the tree */
         if(errorCode == 1)
         {
            printf("%sImpossible d'afficher la notation postfixee de cette arbre.\n%s",
                ROUGE,BLANC);
         }

         p = '!'; /* !  is equal to the head of the tree */
         i = 'm';

         printf("%sIII/ Recherche du noeud pour inserer le nouveau noeud: \n%s",VERT,
             BLANC);

         pere = rechercher(arbre, p, &errorCode);
         /* can't find the node */
         if(errorCode == 0)
         {
            printf("Le nouveau noeud %c sera inseré sur le noeud : %c\n",i,p);
            printf("%sIV/ Insertion du nouveau noeud: \n%s",VERT,BLANC);
            insertNode(pere,i,&errorCode);

            if(errorCode == 0)
            {
               printf("Insertion réussie !\n");
            }
            else
            {
               printf("%sCe noeud est déjà présent dans l'arbre.Insertion annulée. \n%s",
                   ROUGE,BLANC);
            }
         }
         else
         {
            printf("%sImpossible de trouver ce noeud %c, êtes-vous sûr qu'il existe ?\n%s
                ",ROUGE,p,BLANC);
         }

         printf("%sV/ Deuxieme representation Postfixee après insertion: \n%s",VERT,BLANC
             );

         /* display the postfix notation of this tree, startint at vLink because we don't
              want to display the head of the tree */
         repPostFixe(arbre->vLink,&errorCode);

         if(errorCode == 1)
         {
            printf("%sImpossible d'afficher la notation postfixee de cette arbre.\n%s",
                ROUGE,BLANC);
         }

         printf("%sVI/ Copie de l'arbre de base: \n%s",VERT,BLANC);
         newTree = copyTree(arbre,&errorCode);
         if(errorCode == 0)
         {
            printf("Copie réussie !\n");
            printf("%sVII/ troisieme representation postfixee avec la nouvelle structure
                de noeud: \n%s",VERT,BLANC);
```

```
            posFixNotationFather(*newTree);
        }
        else
        {
            printf("%sImpossible de copier cette arbre. Peut-être que votre pile est trop
                petite ?\n%s",ROUGE,BLANC);
        }

        printf("%sVIII/ Libération de l'arbre de base: \n%s",VERT,BLANC);
        freeTree(arbre->vLink,&errorCode);
        if(errorCode == 0)
        {
            printf("Libération réussie !\n");
        }
        else
        {
            printf("%sImpossible de libérer cette arbre. Problème de pile peut-être ?\n%s
                ",ROUGE,BLANC);
        }

        /* Free the last tree */
        printf("%sIX/ Libération de l'arbre modifié.\n%s",VERT,BLANC);
        freeTreeFather(*newTree);
        printf("Libération réussie !\n");
    }
    else
    {
        printf("Votre arbre est vide ! Va falloir le remplir !\n");


    }


}
else
{
    printf("%sImpossible de créer cette arbre. Peut-être que votre pile est trop petite
        ?\n%s",ROUGE,BLANC);
}

free(arbre); /* we finally free the head of the tree */

return(EXIT_SUCCESS);
}
```

2.2 createNode

Principe : createNode (char courant)

on alloue un nouveau noeud
On lui associe courant comme valeur
On fait pointer vers NULL ses autres pointeurs
On renvoie le noeud

FIN

Lexique :

- Paramètre(s) de la fonction
  - courant : caractère associé au nouveau noeud.
- Variables locales:
  - node : contient la structure de nouveau noeud qui sera renvoyé.

Programme Commenté :

```
noeud_t* createNode(char courant)
{
    /* Create a new node */
    noeud_t* node;

    node = (noeud_t*)malloc(sizeof(noeud_t));

    /* set its new value */
    node->value = courant;
    node->hLink = NULL;
    node->vLink = NULL;

    return(node);
}
```

2.3 incrementNbSon

Principe : incrementNbSon

on recupère le dernier élément empiler dans stack
on ajoute +1 à son nombre de fils
on remet cet élément dans la pile

FIN

Lexique :

- Paramètre(s) de la fonction
  - stack : c'est la pile qui contient des noeuds de l'arbre.
  - elmt : permet de recupérer un noeud de la pile.
  - errorCode : indique si l'opération (pop ou push) s'est bien déroulée

Programme Commenté :

```
void incrementNbSon(stack_t* stack, typeStack elmt, int*errorCode)
{
   /* get the node by popping it from the stack */
   pop(stack,&elmt, errorCode);

   /* change its number of son */
   elmt.nb_fils+=1;

   /* push it to its former place */
   push(stack,elmt, errorCode);
}
```

2.4 pushBis

Principe : pushBis

On initialise notre nouvel élément
On met son nombre de fils à 1
On met cet élément dans la pile

FIN

Lexique :

- Paramètre(s) de la fonction
  - stack : c'est la pile qui contient des noeuds de l'arbre.
  - elmt : permet de recupérer un noeud de la pile.
  - errorCode : indique si l'opération (pop ou push) s'est bien déroulée.
  - cur : c'est le noeud que l'on veut empiler.

Programme Commenté :

```
void pushBis(stack_t* stack, typeStack elmt, int* errorCode, noeud_t* cur)
{
   /* function to add a new element to a stack */
   elmt.adr = cur;
   elmt.nb_fils = 1;
   push(stack, elmt, errorCode);

}
```

2.5 repPostFixe

Principe : repPostFixe

On initialise des pointeurs, une pile et des variables;
Si l'arbre est non vide alors:
Tant qu'on n'a pas parcouru tout l'arbre faire:
Si l'élément a déjà été traité alors :
On affiche l'élément avec son nombre de fils;
S'il a un frère alors:
On accède à son frère;
Sinon:
On accède à son père s'il en a un;
Sinon: [S'il n'a pas déjà été traité]
Tant qu'il existe un fils faire :
On accède au fils de l'élément;
On affiche le dernier fils;
S'il a un frère alors:
On accède à son frère;
Sinon: [il n'a pas de frère]
On accède à son père si il existe;

FIN

Lexique :

- Paramètre(s) de la fonction
  - L'adresse de l'arbre
  - L'adresse d'une variable de Code Erreur
- Variables locales:
  - cur est le pointeur courant qui parcourt l'arbre
  - stack est la pile qui servira à stocker les élément lors du parcours et de remonter dans l'arbre
  - elmt est le type d'élément stocké dans la pile stack
  - wasInStack est un entier représentant un booléen afin de savoir si l'élément a déjà été empilé et donc traité
  - end est un entier traduisant un booléen afin de savoir si on a parcouru tout l'arbre

Programme Commenté :

```
void repPostFixe(noeud_t* tree, int* errorCode)
{

   noeud_t*  cur;
   stack_t*  stack;

   typeStack elmt;

   int       end;
   int         wasInStack;
   int       errorCodeStack;

   *errorCode = 0;

   cur = tree;
   stack = initStack(SIZE_STACK, &errorCodeStack);

   end = 1;
   wasInStack = 0;

   /* the stack is well initialized */
   if(errorCodeStack == 1)
   {
      if(cur != NULL) /* if the tree is not empty */
      {
         /* while we don't have gone throught the entire tree */
         while(cur != NULL && end != 0)
         {
            if(wasInStack == 1) /* we have already processed this element */
            {
               printf("(%c,%d)",cur->value,elmt.nb_fils); /* we display the element */

               if(cur->hLink!=NULL) /* Has it a brother ? */
               {
                  cur = cur->hLink ;
                  if (!isStackEmpty(stack)) /* if the stack is not empty */
                  {
                     incrementNbSon(stack,elmt,&errorCodeStack);
```

```
                    if(errorCodeStack != 1) /* stop the algorithm if we had a problem
                        with the stack */
                    {
                        end = 1;
                    }
                }
                wasInStack = 0; /* this element hasn't been processed yet */
            }
            else          /* we pull up the tree if it is possible */
            {
                if(!isStackEmpty(stack))
                {
                    pop(stack,&elmt,&errorCodeStack);

                    if(errorCodeStack != 1) /* stop the algorithm if we had a problem
                        with the stack */
                    {
                        end = 1;
                    }
                    else /* otherwise, we can continue what we were doing */
                    {
                        cur = elmt.adr;
                    }
                }
                else
                {
                    end=0;
                }
            }
        }
        else
        {
            /* while there is a son */
            while(cur != NULL && cur->vLink != NULL)
            {
                pushBis(stack,elmt,&errorCodeStack,cur);

                if(errorCodeStack != 1) /* stop the algorithm if we had a problem with
                    the stack */
                {
                    end = 1;
                }

                cur = cur->vLink; /* the pointer points to its son */
            }

            printf("(%c,0)", cur->value);

            if (cur->hLink != NULL) /*if it has a brother */
            {
                cur = cur->hLink; /* the pointer points to its brother */

                if (!isStackEmpty(stack)) /* if the stack is not empty */
                {
                    incrementNbSon(stack,elmt,&errorCodeStack);

                    if(errorCodeStack != 1) /* stop the algorithm if we had a problem
                        with the stack */
                    {
                        end = 1;
                    }
                }

            }
            else /* if there isn't a brother we pull up the tree if it's possible */
            {
                if (!isStackEmpty(stack)) /* if the stack is not empty we can pull up
                    in the tree */
                {
                    pop(stack,&elmt,&errorCodeStack);

                    if(errorCodeStack != 1) /* stop the algorithm if we had a problem
                        with the stack */
                    {
                        end = 1;
                    }
                    else
                    {
                        cur = elmt.adr; /* the pointer points to its father */
                        wasInStack = 1;
                    }
                }
                else
                {
                    end = 0;
                }
            }
        }
    }
}
freeStack(stack); /* don't forget to dealloc the stack */
```

```
        printf("\n");

    }
    else
    {
        *errorCode = 1; /* failed to initialized stack */
    }

}
```

2.6 createTree

Principe : createTree

On initialise une pile, des pointeurs de parcours, et des variables
Pour chaque caractère de notre chaine :
    Si le caractère courant est une parenthèse ouvrante:
        On le sauvegarde pour la prochaine itération
    Si le caractère est une parenthèse fermante :
        Si la pile est non vide:
            On la dépile pour faire pointer le noeud de parcours sur un noeud père
        Sinon:
            On s'arrête
    Si le caractère est une virgule:
        On le sauvegarde pour la prochaine itération
    Sinon [Si c'est une lettre]:
        On ajoute un noeud au noeud courant : sur le le lien vertical si le caracètre pécèdent était une parenthèse ouvrante, sur le lien horizontale si le caractère précèdent était une virgule ou une parenthèse fermante.
On renvoie l'arbre
Si il y a eu un problème lors de la création
    On libère l'arbre !

FIN

Lexique :

- Paramètre(s) de la fonction
  - treeString contient la chaîne de caractère décrivant l'arbre à créer.
- Variable(s) locale(s)
  - curr est le pointeur courant qui parcourt la liste.

Programme commenté :

```
noeud_t* createTree(char *treeString,int* errorCode)
{
    /* Init our Stack */
    stack_t* stack;
    int    errorCodeStack;
    int    size;

    noeud_t*  head = malloc(sizeof(noeud_t)); /* create the head of the tree */
    noeud_t* prec;

    char     courant;
    int      index; /* index will go throught all the char* format */
    int      parentOpen; /*indicate if the previous character was a parenthesis */
    int    fini;

    size = strlen(treeString);

    head->value = '!';
    head->vLink = NULL;
    head->hLink = NULL;
    prec = head;
    index = 0;
    parentOpen = 0;
    stack = initStack(SIZE_STACK,&errorCodeStack);
    fini = 0;
    *errorCode = 0;

    if(errorCodeStack == 1)
    {
        while(index < size && fini == 0)
        {
            courant = treeString[index]; /* get the new character to process */

            switch(courant)
            {
```

```
            case 40: /* "(" */
            {
               parentOpen = 1;
               break;
            }
            case 41: /* ")" */
            {
               if(!isStackEmpty(stack)) /* If the stack is not empty */
               {
                  T_elmtPile elmtPile;

                  pop(stack,&elmtPile,&errorCodeStack);
                  if(errorCodeStack == 0)
                  {
                     fini = 1;
                     *errorCode = 1;
                  }
                  else
                  {
                     prec = elmtPile.adr;
                  }
               }
               else
               {
                  fini = 1;
               }

               break;
            }
            case 44: /* "," */
            {
               parentOpen = 0;
               break;
            }
            default:
            {
               /* Creation of the node */
               noeud_t*   node;
               T_elmtPile elmtPile;

               node = createNode(courant);

               if(parentOpen == 1)
               {
                  prec->vLink = node;

                  elmtPile.adr = prec;
                  elmtPile.nb_fils = 0;

                  push(stack, elmtPile , &errorCodeStack);
                  if(errorCodeStack == 0) /* problem during push, we need to stop right
                      here */
                  {
                     fini = 1;
                     *errorCode = 1;
                  }

               }
               else /* We have a comma or a closed parenthesis */
               {
                  /* Create the vertical link */
                  prec->hLink = node;
               }

               /* move the pointer to the next node */
               prec = node;

               parentOpen = 0;
               break;
            }


         }
         index++;
      }
      freeStack(stack);
   }
   else
   {
      *errorCode = 1;
   }

   /* if the creation went wrong, we need to free the partial tree here */
   if(*errorCode == 1)
   {
      freeTree(head->vLink,&errorCodeStack);
   }

   return(head);
}
```

2.7 rechercher

Principe : rechercher

On initialise des pointeurs et des variables;
Si l'arbre est non Vide alors:
Tant qu'on a pas parcouru tout l'arbre ou qu'on n'a pas trouvé la valeur v faire:
Si l'élément a un fils alors:
On stock l'élément dans la file;
Si l'élément possède un frère alors:
on accède à son frère;
Sinon:
Si la file est non vide alors:
On retourne sur le premier élément enfilé;
Sinon:
On a parcouru tout l'arbre, on s'arrête;

FIN

Lexique :

- Paramètre(s) de la fonction
  - L'adresse de l'arbre
  - L'adresse d'une variable de Code Erreur
  - La valeur v du nœud à chercher
- Variables locales:
  - cur est le pointeur courant qui parcourt l'arbre
  - queue est la file qui servira à stocker les élément lors du parcours de l'arbre selon le 1er ordre
  - end est un entier traduisant booléen

Programme Commenté :

```
noeud_t *  rechercher(noeud_t * tree, char v, int * errorCode)
{
   int       errorCodeQueue; /* indicates if we had a problem with the queue */
   queue_t*  file;
   noeud_t*  cur;
   queueType elmt;
   int     end;

   *errorCode = 0;

   /* init a queue */
   file = initQueue(SIZE_STACK,&errorCodeQueue);
   cur = tree;
   end = 0;

   /* if the initialization of the queue went good */
   if(errorCodeQueue == 1)
   {
      /* if the three is not empty */
      if (cur != NULL)
      {
         while ( cur != NULL && cur->value != v && end == 0) /* while we don't have found
              the value v or we don't have gone through the entire tree */
         {

            if (cur->vLink != NULL) /* if it has a son */
            {

               elmt.adr = cur; /* we stock the current element then we pull it on the
                   queue */
               enterQueue(file,elmt,&errorCodeQueue);

               /* we had a problem with the queue -> exit */
               if(errorCodeQueue != 1)
               {
                  *errorCode = 1;
                  end = 1;
               }
            }
            if (cur->hLink != NULL && end == 0) /* if it has a brother */
            {
               cur = cur->hLink; /* the pointer points to its brother */
            }
            else /* if it doesn't have a brother */
            {

               if(!isQueueEmpty(file))
               {
```

```
                    cur = file->base->adr; /* we go back on the first element threaded */
                    elmt = leaveQueue(file,&errorCodeQueue);

                    /* we had a problem with the queue -> exit */
                    if(errorCodeQueue != 1)
                    {
                        *errorCode = 1;
                        end = 1;
                    }
                    else
                    {
                        cur = cur->vLink; /* the pointer points to its son */
                    }
                }
                else
                {
                    end = 1;
                    *errorCode = 1;
                }

            }
        }
    }

    freeQueue(file);
}
else
{
    *errorCode = 1;
}

return cur;
}
```

2.8 createNodeForInsertion

Principe : createNodeForInsertion

On crée une structure qu'on alloue
On lui associe un caractère w.
On lui associe un lien horizontal
Le reste pointe vers NULL.
On renvoie cette structure.

FIN

Lexique :

- Paramètre(s) de la fonction
  - w : c'est le caractère associé au nouveau noeud.
  - horizontal : c'est le lien horizontal du nouveau noeud. Il peut-être NULL.
- Variables locales:
  - temp : pointeurs vers la nouvelle structure crée.

Programme Commenté :

```
noeud_t* createNodeForInsertion(char w,noeud_t *horizontal)
{
    /* Create a node for the insertNode function */
    noeud_t* temp;
    temp =  (noeud_t *)malloc(sizeof(noeud_t));

    temp->value = w;
    temp->hLink = horizontal;
    temp->vLink = NULL;

    return(temp);
}
```

2.9 insertNode

Principe : insertNode

On initialise des pointeurs et des variables;
Si le père existe:
S'il n'a pas de fils alors :
On insère en tête le fils;
Sinon: [il a au moins un fils]
Si le premier fils est avant que le fils à insérer dans l'ordre alphabétique alors:
Tant qu'on n'a pas trouvé la place où on insère le fils faire:
On parcourt ses frères
Si on est à la fin des fils alors:
On insère le fils à la fin des fils;
Sinon: [on a trouvé une place où insérer le fils entre 2 frères]
Si le fils n'existe pas déjà alors:
On insère le fils;
Sinon: [le fils existe déjà pas besoin]
On met fin au parcours;
Sinon: [alors on l'insère en tête]
On insère le fils;

FIN

Lexique :

- Paramètre(s) de la fonction
  - L'adresse du nœud où l'on doit insérer le fils
  - L'adresse d'une variable de Code Erreur
  - La valeur w du fils à insérer
- Variables locales:
  - cur est le pointeur courant qui parcourt l'arbre
  - prec est le pointeur qui pointe sur l'élément précédent, ici sur le frère précédent

Programme Commenté :

```
void insertNode(noeud_t* tree,char w, int* errorCode)
{
   /* declares some variable to go through the tree */
   noeud_t* cur;
   noeud_t* prec;

   cur = tree;
   *errorCode = 0;
   prec = NULL;

   /*if the subtree exists */
   if (cur != NULL)
   {
      if (cur->vLink == NULL)  /* so there isn't son */
      {
         cur->vLink= createNodeForInsertion(w,NULL);
         cur = cur->vLink;

      }
      else /* then he has at least one son */
      {
         prec = cur;
         cur = cur->vLink;
         if (cur->value < w)   /* Insert the son in the alphabetical order */
         {
            while(cur != NULL && cur->value <w) /* We search where to insert the son */
            {
               prec = cur;
               cur = cur->hLink;
            }
            if(cur == NULL) /*went through all the sons, we can insert it at the end */
            {
               prec->hLink = createNodeForInsertion(w,NULL);

            }
            else     /*we insert the element in the middle of its brothers*/
            {
               if (cur->value != w)  /*the element hasn't created yet*/
               {
                  prec->hLink = createNodeForInsertion(w,cur); /* link it to the next
                      brother */
               }
               else /* the element is already created so no need to insert it*/
               {
```

```
                    *errorCode = 1;
                }

            }
        }
        else  /*if the element is before the first son in the alphabet */
        {
            if (cur->value > w) /* so we insert the element into the head */
            {
                prec->vLink = createNodeForInsertion(w,cur); /* link it to the next
                    brother */
            }
            else   /* the first son is already the element, no need to insert it */
            {
                *errorCode = 1;
            }
        }

    }
  }

}
```

2.10 createModifiedNode

Principe : createModifiedNode

On crée un nouveau noeud modifié.
On lui associe le caractère du noeud curr
Les autres pointeurs sont initialisés à NULL;
La valeur papa du noeud est initialisée à NULL.
On renvoie le nouveau noeud.

FIN

Lexique :

- Paramètre(s) de la fonction
  - cur : c'est le noeud qui donne la valeur à notre nouveau noeud.
  - pere : le pointeur papa de nouvelle structure pointera vers le noeud pere
- Variables locales:
  - node : c'est le pointeur vers notre nouvelle structure

Programme Commenté :

```
newNode_t* createModifiedNode(noeud_t* cur,newNode_t* pere)
{
   newNode_t* node = malloc(sizeof(newNode_t));

   node->value = cur->value;
   node->vLink = NULL;
   node->hLink = NULL;

   node->papa = pere;

   return(node);
}
```

2.11 copyTree

Principe : copyTree

On initialise une pile, la tête du nouvel arbre, et des variables de parcours/contrôle.
Si l'arbre est non-vide :
Tant qu'on a pas parcouru l'arbre ou rencontré une erreur:
Tant que le noeud courant possède un lien vertical et que c'est la première fois qu'on le visite
On crée un noeud modifié à partir du noeud courant/
On empile ce noeud dans la pile.
On fait descendre le pointeur courant et le père dans la hiérarchie.
Si le noeud courant possède un frère :
On crée un noeud modifié à partir du noeud courant.
On fait pointer le noeud courant vers son frère.
Sinon
Si la pile est non vide
On récupère le dernier élément de la pile.
On modifie les pointeurs de parcours du nouvel arbre grâce au père du noeud courant.
Sinon
On s'arrête.
On renvoie le nouvel arbre.
FIN

Lexique :

- Paramètre(s) de la fonction
  - tree : c'est le pointeur de l'arbre que l'on veut copier
  - errorCode : indique si la fonction s'est bien déroulée
- Variables locales:
  - newTree: c'est le pointeur vers le nouvel arbre.
  - end: contrôle la terminaison de l'algorithme.
  - wasInStack: indique si le noeud courant a déjà été parcouru.
  - cur: est le pointeur courant qui parcourt l'arbre de base.
  - stack: c'est le pointeur de la pile.
  - currNewTree: c'est le pointeur de parcours du nouvel arbre.
  - father : pointeur de parcours qui pointe sur le père du pointeur courant.
  - elmt: variable qui permet d'empiler un noeud dans la pile.
  - errorCodeStack : indique si il y a eu un problème avec la pile.

Programme Commenté :

```
newNode_t* copyTree(noeud_t* tree,int *errorCode)
{
   newNode_t*  newTree;
   int      end;
   int         wasInStack; /* is the previous node was popped from the stack */
   noeud_t*    cur;
   stack_t*    stack;
   newNode_t*  currNewTree;
   newNode_t*  father; /* keep track of the father of the current node */
   typeStack   elmt;
   int       errorCodeStack;

   cur = tree;
   end = 1;
   wasInStack = 0;

   newTree = malloc(sizeof(newNode_t));
   newTree->papa = NULL;
   newTree->hLink = NULL;
   newTree->vLink = NULL;

   currNewTree = newTree;
   father = currNewTree;

   *errorCode = 0;

   /* init a stack */
   stack = initStack(SIZE_STACK, &errorCodeStack);

   if(errorCodeStack == 1)
   {
```

```
    if(cur->vLink != NULL) /* if the tree is not empty */
    {
       /* while we don't have gone throught the entire tree */
       while(cur != NULL && end == 1)
       {
          /* make sure the next node has a vertical link and the curr was not pop from
              the stack */
          if(cur->vLink != NULL && wasInStack == 0)
          {
             /* while there is a son */
             do
             {
                cur = cur->vLink;

                currNewTree->vLink = createModifiedNode(cur,father);
                currNewTree = currNewTree->vLink;

                /* if the node hasn't vLink, then no need to push it ! */
                if(cur->vLink != NULL)
                {
                   pushBis(stack,elmt,&errorCodeStack,cur);
                   if(errorCodeStack != 1)
                   {
                      *errorCode = 1;
                      end = 0; /* end the algorithm */
                   }
                   else
                   {
                      father = currNewTree;
                   }
                }
             }
             while(cur != NULL && cur->vLink != NULL && wasInStack == 0);
          }
          wasInStack = 0;


          if (cur->hLink != NULL && end == 1) /*if it has a brother */
          {
             cur = cur->hLink;
             currNewTree->hLink = createModifiedNode(cur,father);
             currNewTree = currNewTree->hLink;

             /* if the node hasn't vLink, then no need to push it ! */
             if(cur->vLink != NULL)
             {
                pushBis(stack,elmt,&errorCodeStack,cur);

                if(errorCodeStack != 1)
                {
                   *errorCode = 1;
                   end = 0; /* end the algorithm */
                }
                else
                {
                   father = currNewTree;
                }
             }
          }
          else /* if there isn't a brother we pull up the tree if it's possible */
          {
             /* If the stack is not empty */
             if(!isStackEmpty(stack))
             {
                pop(stack,&elmt,&errorCodeStack);

                if(errorCodeStack != 1)
                {
                   *errorCode = 1;
                   end = 0; /* end the algorithm */
                }
                else
                {
                   cur = (elmt.adr);
                   wasInStack = 1;

                   currNewTree = currNewTree->papa; /* came back to the father */
                   father = currNewTree->papa;
                }
             }

             if(isStackEmpty(stack) && cur->hLink == NULL && end == 1)
             {
                end = 0;
             }
          }
       }
       freeStack(stack);
    }
    else
```

```
    {
        *errorCode = 1;
    }

    return(newTree);
}
```

2.12 posFixNotationFather

Principe : posFixNotationFather

On initialise une pile et des variables de parcours/contrôle.
Si l'arbre est non-vide :
Tant qu'on a pas parcouru l'arbre ou rencontré une erreur:
Tant que le pointeur courant à un fils et qu'il n'a pas déjà été parcouru
On descend le noeud courant dans l'arbre.
Si le noeud courant n'a pas été déjà visité:
On affiche le noeud courant.
Si le noeud courant possède un frère :
On parcourt via le lien horizontal
Sinon
Tant que le père du noeud courant est non-null ou que le noeud courant a un lien horizontal null
On remonte dans l'arbre avec le lien du père.
On affiche le noeud courant.
Si on a atteint la tête de l'arbre.
on s'arrête.

FIN

Lexique :

- Paramètre(s) de la fonction
  - tree : c'est le pointeur de l'arbre que l'on veut afficher
- Variables locales:
  - cur: est le pointeur courant qui parcourt l'arbre de base.
  - end: contrôle la terminaison de l'algorithme.
  - backFromFather: indique si le noeud courant a déjà été parcouru.

Programme Commenté :

```
void posFixNotationFather(newNode_t tree)
{
    newNode_t*  cur = tree.vLink;

    /* some variable to stop and control the algorithm */
    int        end;

    /* backFromFather prevent the algorithm to process node that it has already processed
       */
    int      backFromFather;

    end = 0;
    backFromFather = 0;

    if(cur != NULL) /* if the tree is not empty */
    {
        /* while we don't have gone throught the entire tree */
        while(cur  != NULL && end == 0)
        {

            /* while there is a son, we process it */
            while(cur != NULL && cur->vLink != NULL && backFromFather == 0)
            {
                cur = cur->vLink; /* we go down throught the vertical link */
            }

            /* if this node had already been processed, we don't display it again */
            if(backFromFather == 0)
            {
                printf("| %c | ", cur->value);
            }
            backFromFather = 0;

            /*if it has a brother */
            if (cur->hLink != NULL)
            {
                cur = cur->hLink; /* then we use the horizontal link */
            }
```

```
            else /* if there isn't a brother we come back to the father of the nodes */
            {
               while (cur->papa != NULL && cur->hLink == NULL) /* wait for a father with a
                   horizontal link */
               {
                  backFromFather = 1; /* indicates that we will not process this node later
                      */
                  cur = cur->papa;

                  if(cur->papa != NULL) /* if we reached the head of the tree, we don't
                      display it */
                  {
                     printf("| %c | ", cur->value);
                  }
               }

               if(cur->papa == NULL) /* end the algorithm if we hit the head */
               {
                  end = 1;
               }
            }
         }
      }
      printf("\n");
}
```

2.13 freeTreeFather

Principe : freeTreeFather

On initialise des variables de parcours/contrôle.
Si l'arbre est non-vide :
Tant qu'on a pas parcouru tout l'arbre ou rencontré un problème:
Tant que le noeud courant a un fils et qu'il n'a pas déjà été visité
on descend dans la hiérarchie de l'arbre.
Si le noeud courant n'a pas déjà été visité
On enregistre le noeud courant
Si le noeud courant a un frère
Si le noeud a déjà été visité
On enregistre le noeud courant
On parcourt via le lien horizontal
On libère le noeud enregistré
Sinon
Tant que le père du noeud courant n'est pas NULL et que le frère du noeud courant est NULL
Si le noeud courant a déjà été visité
On enregistre le noeud courant
On revient en arrière grâce au père du noeud courant
On libère le noeud enregistré
Si on a atteint la tête de l'arbre
On s'arrête.
On libère le noeud enregistré.
FIN

Lexique :

- Paramètre(s) de la fonction
  - tree : c'est le pointeur de l'arbre que l'on veut libérer
- Variables locales:
  - cur: est le pointeur courant qui parcourt l'arbre.
  - end: contrôle la terminaison de l'algorithme.
  - backFromFather: indique si le noeud courant a déjà été parcouru.
  - temp: cette variable enregistre les noeuds courant à libérer.

Programme Commenté :

```
void freeTreeFather(newNode_t tree)
{
   newNode_t*  cur;

   /* some variable to stop and control the algorithm */
   int         end;

   /* backFromFather prevent the algorithm to process node that it has already processed
       */
   int       backFromFather;
```

```
    newNode_t* temp;

    end = 0;
    backFromFather = 0;
    cur = tree.vLink;

    if(cur != NULL) /* if the tree is not empty */
    {
        while(cur  != NULL && end == 0)/* while we don't have gone throught the entire tree
             */
        {
            /* while there is a son, we process it */
            while(cur != NULL && cur->vLink != NULL && backFromFather == 0)
            {
                cur = cur->vLink; /* we go down throught the vertical link */
            }

            /* save the current node that will be delete */
            if(backFromFather == 0)
            {
                temp = cur;
            }

            /*if it has a brother */
            if (cur->hLink != NULL)
            {
                /* now we can delete this node, because we already went through its sons */
                if(backFromFather == 1)
                {
                    temp = cur;
                    backFromFather = 0;
                }

                cur = cur->hLink; /* then we use the horizontal link */

                free(temp);

            }
            else /* if there isn't a brother we come back to the father of the nodes */
            {
                while (cur->papa != NULL && cur->hLink == NULL) /* wait for a father with a
                     horizontal link */
                {
                    /* each time we go up, we have to delete node */
                    if(backFromFather == 1)
                    {
                        temp = cur;
                    }
                    cur = cur->papa;

                    free(temp);
                    backFromFather = 1; /* indicates that we will delete this node in the next
                         loop */
                }

                if(cur->papa == NULL) /* end the algorithm if we hit the head */
                {
                    end = 1;
                }
            }


        }
    }

    free(cur);
}
```

2.14 freeTree

Principe : freeTree

On initialise une pile et des variables de parcours/contrôle.
Si l'arbre est non-vide :
Si l'élément courant a déjà été parcouru:
Tant qu'on a pas parcouru tout l'arbre ou rencontré un problème:
On enregistre le noeud courant.
Si le noeud courant possède un frère:
On parcours via le lien horizontal
Sinon
Si la pile est non vide:
On récupère le dernier élément empilé.
Sinon :
On s'arrête.
Sinon:
Tant que le noeud courant a un fils:
on empile le noeud courant.
On passe à son fils.
On enregistre le noeud courant.
Si le noeud courant a un frère:
On parcourt via le lien horizontal.
Sinon:
Si la pile est non-vide:
on récupère le dernier élément empilé.
Sinon:
On s'arrête.
On libère le noeud enregistré.
On libère la pile.
FIN

Lexique :

- Paramètre(s) de la fonction
  - tree : c'est le pointeur de l'arbre que l'on veut libérer
- Variables locales:
  - cur: est le pointeur courant qui parcourt l'arbre.
  - stack: pile contenant des noeuds de l'arbre à supprimer.
  - elmt: variable qui permet d'empiler des noeuds.
  - end: contrôle la terminaison de l'algorithme.
  - wasInStack: indique si le noeud courant a déjà été parcouru.
  - errorCodeStack: indique si il y a eu un problème avec la pile.

Programme Commenté :

```
void freeTree(noeud_t* tree,int *errorCode)
{
   noeud_t*  cur;
   stack_t*  stack;
   typeStack elmt;
   int       end;
   int        wasInStack;
   int     errorCodeStack;

   *errorCode = 0;
   /* here, we don't give the choice to the user for the size of the stack */
   stack = initStack(100, &errorCodeStack);
   cur = tree;
   end = 1;
   wasInStack = 0;

   /* if we had a problem during the initialization of the stack */
   if(errorCodeStack == 1)
   {
      if(cur != NULL) /* if the tree is not empty */
      {
         /* while we don't have gone throught the entire tree */
         while(cur != NULL && end != 0)
         {
            if(wasInStack == 1) /* we have already processed this element */
            {
               noeud_t* temp;
               temp = cur;
```

```
            if(cur->hLink!=NULL) /* Has it a brother ? */
            {
               cur = cur->hLink ;

               wasInStack = 0; /* this element hasn't been processed yet */
            }
            else /* we pull up the tree if it is possible */
            {
               if(!isStackEmpty(stack))
               {
                  pop(stack,&elmt,&errorCodeStack);

                  /* if we had a problem with the stack */
                  if(errorCodeStack != 1)
                  {
                     *errorCode = 1;
                     end = 0; /* end the algortihm */
                  }
                  else
                  {
                     cur = elmt.adr;
                  }
               }
               else
               {
                  end = 0;
               }
            }

            free(temp);

         }
         else
         {
            noeud_t* temp;

            /* while there is a son */
            while(cur != NULL && cur->vLink != NULL && end == 1)
            {
               pushBis(stack,elmt,&errorCodeStack,cur);

               /* if we had a problem with the stack */
               if(errorCodeStack != 1)
               {
                  *errorCode = 1;
                  end = 0; /* end the algortihm */
               }
               else
               {
                  cur = cur->vLink; /* the pointer points to its son */
               }
            }

            temp = cur;

            if (cur->hLink != NULL) /*if it has a brother */
            {
               cur = cur->hLink; /* the pointer points to its brother */
            }
            else /* if there isn't a brother we pull up the tree if it's possible */
            {
               if (!isStackEmpty(stack)) /* if the stack is not empty we can pull up
                   in the tree */
               {
                  pop(stack,&elmt,&errorCodeStack);

                  /* if we had a problem with the stack */
                  if(errorCodeStack != 1)
                  {
                     *errorCode = 1;
                     end = 0; /* end the algortihm */
                  }
                  else
                  {
                     cur = elmt.adr; /* the pointer points to its father */

                     wasInStack = 1;
                  }
               }
               else
               {
                  end = 0;
               }
            }

            free(temp);

         }
      }
   }
```

```
        freeStack(stack); /* don't forget to free the stack even if we had a problem */
    }
    else
    {
        *errorCode = 1;
    }
}
```

3 Compte rendu d'exécution

3.1 Cas : Cas général

treeString = (a(b(k(h,u)z)f(m)x(p,v,w)))
SIZE_STACK = 50
noeud d'insertion = b
noeud à insérer = l

Sortie:

```
==19896== Memcheck, a memory error detector
==19896== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==19896== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19896== Command: ./bin/main_program
==19896==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
III/ Recherche du noeud pour inserer le nouveau noeud:
Le nouveau noeud l sera inseré sur le noeud : b
IV/ Insertion du nouveau noeud:
Insertion réussie !
V/ Deuxieme representation Postfixee après insertion:
(h,0)(u,0)(k,2)(l,0)(z,0)(b,3)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| h | | u | | k | | l | | z | | b | | m | | f | | p | | v | | w | | x | | a |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==19896==
==19896== HEAP SUMMARY:
==19896==     in use at exit: 0 bytes in 0 blocks
==19896==   total heap usage: 41 allocs, 41 frees, 7,112 bytes allocated
==19896==
==19896== All heap blocks were freed -- no leaks are possible
==19896==
==19896== For counts of detected and suppressed errors, rerun with: -v
==19896== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Cas général

3.2 Cas : 2 forêts

treeString = (a(b(k(u,w)))c(l(j,x)m(d,e)))
SIZE_STACK = 50
noeud d'insertion = a
noeud à insérer = l

Sortie:

```
==20220== Memcheck, a memory error detector
==20220== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20220== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20220== Command: ./bin/main_program
==20220==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(u,0)(w,0)(k,2)(b,1)(a,1)(j,0)(x,0)(l,2)(d,0)(e,0)(m,2)(c,2)
III/ Recherche du noeud pour inserer le nouveau noeud:
Le nouveau noeud l sera inseré sur le noeud : a
IV/ Insertion du nouveau noeud:
Insertion réussie !
V/ Deuxieme representation Postfixee après insertion:
(u,0)(w,0)(k,2)(b,1)(l,0)(a,2)(j,0)(x,0)(l,2)(d,0)(e,0)(m,2)(c,2)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| u | | w | | k | | b | | l | | a | | j | | x | | l | | d | | e | | m | | c |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==20220==
==20220== HEAP SUMMARY:
==20220==     in use at exit: 0 bytes in 0 blocks
==20220==   total heap usage: 41 allocs, 41 frees, 10,712 bytes allocated
==20220==
==20220== All heap blocks were freed -- no leaks are possible
==20220==
==20220== For counts of detected and suppressed errors, rerun with: -v
==20220== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
jeremy@chappie:~/Documents/ISIMA/TP_SDD_ISIMA/TP3$
```

2 forêts

3.3 Cas : Arbre vertical

treeString = (a(b(k)))
SIZE_STACK = 50
noeud d'insertion = a
noeud à insérer = l

Sortie:

```
==20284== Memcheck, a memory error detector
==20284== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20284== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20284== Command: ./bin/main_program
==20284==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(k,0)(b,1)(a,1)
III/ Recherche du noeud pour inserer le nouveau noeud:
Le nouveau noeud l sera inseré sur le noeud : a
IV/ Insertion du nouveau noeud:
Insertion réussie !
V/ Deuxieme representation Postfixee après insertion:
(k,0)(b,1)(l,0)(a,2)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| k | | b | | l | | a |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==20284==
==20284== HEAP SUMMARY:
==20284==     in use at exit: 0 bytes in 0 blocks
==20284==   total heap usage: 23 allocs, 23 frees, 10,208 bytes allocated
==20284==
==20284== All heap blocks were freed -- no leaks are possible
==20284==
==20284== For counts of detected and suppressed errors, rerun with: -v
==20284== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Arbre Vertical

3.4 Cas : Arbre horizontal

treeString = (a,b,c)
SIZE_STACK = 50

noeud d'insertion = a
noeud à insérer = l

Sortie:

```
==20400== Memcheck, a memory error detector
==20400== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20400== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20400== Command: ./bin/main_program
==20400==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(a,0)(b,0)(c,0)
III/ Recherche du noeud pour inserer le nouveau noeud:
Le nouveau noeud l sera inseré sur le noeud : a
IV/ Insertion du nouveau noeud:
Insertion réussie !
V/ Deuxieme representation Postfixee après insertion:
(l,0)(a,1)(b,0)(c,0)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| l | | a | | b | | c |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==20400==
==20400== HEAP SUMMARY:
==20400==     in use at exit: 0 bytes in 0 blocks
==20400==   total heap usage: 23 allocs, 23 frees, 10,208 bytes allocated
==20400==
==20400== All heap blocks were freed -- no leaks are possible
==20400==
==20400== For counts of detected and suppressed errors, rerun with: -v
==20400== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Arbre horizontal

3.5 Cas : Le noeud d'insertion n'existe pas

treeString = (a(b(k(h,u)z)f(m)x(p,v,w)))
SIZE_STACK = 50
noeud d'insertion = g
noeud à insérer = l

Sortie:

```
==20475== Memcheck, a memory error detector
==20475== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20475== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20475== Command: ./bin/main_program
==20475==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
III/ Recherche du noeud pour inserer le nouveau noeud:
Impossible de trouver ce noeud g, êtes-vous sûr qu'il existe ?
V/ Deuxieme representation Postfixee après insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| h | | u | | k | | z | | b | | m | | f | | p | | v | | w | | x | | a |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==20475==
==20475== HEAP SUMMARY:
==20475==     in use at exit: 0 bytes in 0 blocks
==20475==   total heap usage: 39 allocs, 39 frees, 10,656 bytes allocated
==20475==
==20475== All heap blocks were freed -- no leaks are possible
==20475==
==20475== For counts of detected and suppressed errors, rerun with: -v
==20475== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Le noeud d'insertion n'existe pas

3.6 Cas : Insertion du noeud sur la tête de l'arbre

treeString = (a(b(k(h,u)z)f(m)x(p,v,w)))
SIZE_STACK = 50
noeud d'insertion = ! (le caractère '!' est reservée pour identifier la tête)
noeud à insérer = l

Sortie:

```
==20541== Memcheck, a memory error detector
==20541== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20541== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20541== Command: ./bin/main_program
==20541==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
III/ Recherche du noeud pour inserer le nouveau noeud:
Le nouveau noeud l sera inseré sur le noeud : !
IV/ Insertion du nouveau noeud:
Insertion réussie !
V/ Deuxieme representation Postfixee après insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)(l,0)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| h | | u | | k | | z | | b | | m | | f | | p | | v | | w | | x | | a | | l |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==20541==
==20541== HEAP SUMMARY:
==20541==     in use at exit: 0 bytes in 0 blocks
==20541==   total heap usage: 41 allocs, 41 frees, 10,712 bytes allocated
==20541==
==20541== All heap blocks were freed -- no leaks are possible
==20541==
==20541== For counts of detected and suppressed errors, rerun with: -v
==20541== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Insertion du noeud sur la tête de l'arbre

3.7 Cas : Arbre vide

treeString = ()
SIZE_STACK = 50
noeud d'insertion = a
noeud à insérer = l

Sortie:

```
==19965== Memcheck, a memory error detector
==19965== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==19965== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==19965== Command: ./bin/main_program
==19965==
I/ Creation de l'arbre !
Votre arbre est vide ! Va falloir le remplir !
==19965==
==19965== HEAP SUMMARY:
==19965==     in use at exit: 0 bytes in 0 blocks
==19965==   total heap usage: 4 allocs, 4 frees, 1,864 bytes allocated
==19965==
==19965== All heap blocks were freed -- no leaks are possible
==19965==
==19965== For counts of detected and suppressed errors, rerun with: -v
==19965== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Arbre vide

3.8 Cas : Le noeud est déjà présent sur le noeud père

treeString = (a(b(k(h,u)z)f(m)x(p,v,w)))
SIZE_STACK = 50
noeud d'insertion = k
noeud à insérer = h

Sortie:

```
==20601== Memcheck, a memory error detector
==20601== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20601== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20601== Command: ./bin/main_program
==20601==
I/ Creation de l'arbre !
Creation réussie !
II/ Premiere representation Postfixee avant insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
III/ Recherche du noeud pour inserer le nouveau noeud:
Le nouveau noeud h sera inseré sur le noeud : k
IV/ Insertion du nouveau noeud:
Ce noeud est déjà présent dans l'arbre.Insertion annulée.
V/ Deuxieme representation Postfixee après insertion:
(h,0)(u,0)(k,2)(z,0)(b,2)(m,0)(f,1)(p,0)(v,0)(w,0)(x,3)(a,3)
VI/ Copie de l'arbre de base:
Copie réussie !
VII/ troisieme representation postfixee avec la nouvelle structure de noeud:
| h | | u | | k | | z | | b | | m | | f | | p | | v | | w | | x | | a |
VIII/ Libération de l'arbre de base:
Libération réussie !
IX/ Libération de l'arbre modifié.
Libération réussie !
==20601==
==20601== HEAP SUMMARY:
==20601==     in use at exit: 0 bytes in 0 blocks
==20601==   total heap usage: 39 allocs, 39 frees, 10,656 bytes allocated
==20601==
==20601== All heap blocks were freed -- no leaks are possible
==20601==
==20601== For counts of detected and suppressed errors, rerun with: -v
==20601== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Le noeud est déjà présent sur le noeud père

3.9 Cas : Stack de taille 2

treeString = (a(b(k(h,u)z)f(m)x(p,v,w)))
SIZE_STACK = 2
noeud d'insertion = b
noeud à insérer = l

Sortie:

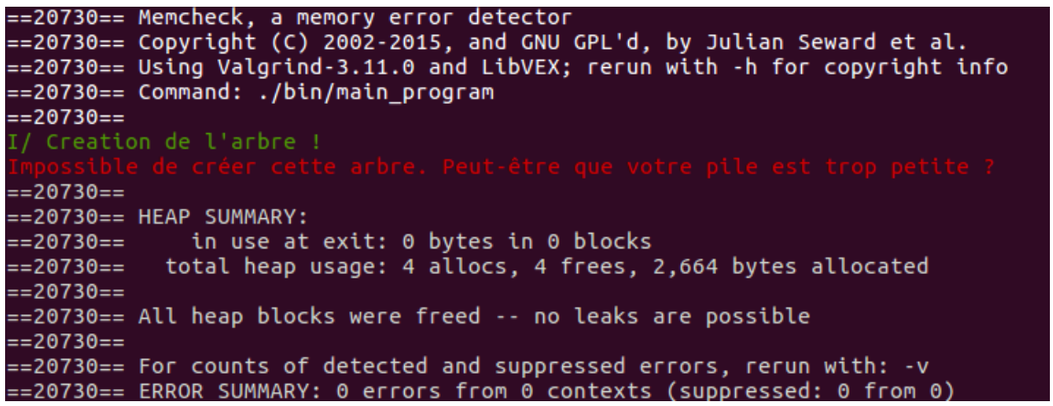
```
==20669== Memcheck, a memory error detector
==20669== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20669== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20669== Command: ./bin/main_program
==20669==
I/ Creation de l'arbre !
Impossible de créer cette arbre. Peut-être que votre pile est trop petite ?
==20669==
==20669== HEAP SUMMARY:
==20669==     in use at exit: 0 bytes in 0 blocks
==20669==   total heap usage: 9 allocs, 9 frees, 2,784 bytes allocated
==20669==
==20669== All heap blocks were freed -- no leaks are possible
==20669==
==20669== For counts of detected and suppressed errors, rerun with: -v
==20669== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Stack de taille 2

3.10 Cas : Stack de taille 0

treeString = (a(b(k(h,u)z)f(m)x(p,v,w)))
SIZE_STACK = 0
noeud d'insertion = b
noeud à insérer = l

Sortie:



Stack de taille 0