DI NARDO VALENTIN
VIELLARD BRUNO

TP 3 de Structure de données :

Gestion d'un dictionnaire arborescent

Table des matières

Présentation :

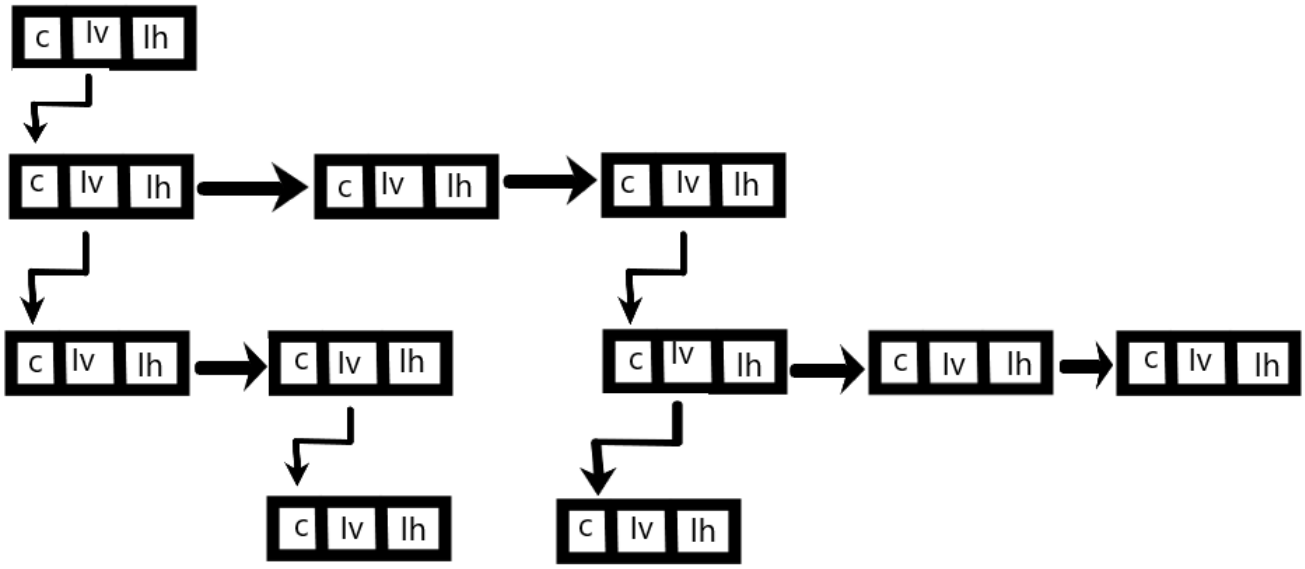
L'objectif de ce TP est la création d'une bibliothèque de gestion d'un dictionnaire arborescent où chaque nœud est constitué d'une lettre et d'un lien vers son frère et un fils. Un dictionnaire est un ensemble de mots. Cette arborescence est représentée par liens verticaux et liens horizontaux qui permettent en suivant ces liens de représenter un mot ou retrouver un mot dont la fin est indiquée par une majuscule.

Une pile est nécessaire à la réalisation de cette bibliothèque.

Structure de Données Utilisées :

ARBRE :

Schéma :



c : variable de type char, elle correspond à une lettre
lv : lien vertical/ lien fils, c'est un pointeur sur un nœud
lh : lien horizontal / lien vers le frère, c'est un pointeur sur un nœud

Si un nœud n'a pas de frère ou de fils lv/lh est égale à NULL, il ne pointe sur rien.

Algorithmes de Principe :

CONSTRUCTION D'UN DICTIONNAIRE

Construction d'un dictionnaire (représentation par lien verticale et lien horizontal) de quelques mots, à partir de la notation algébrique (**supposée sans erreur et comportant les majuscules en fin de mot**).

Lexique :
Expression : L'expression algébrique à lire afin de créer l'arbre
Pile : Une pile afin de conserver les nœuds pour y revenir plus tard
Pointeur courant : Pointeur utiliser pour réaliser les actions (empiler, dépiler, créer un nouveau nœud)
Prec : Double pointeur qui renvoie sur l'adresse de l'arbre (celui que l'on construit)
Stack : Booléen indiquant si à la prochaine itération nous devons empiler ou non

Position : entier utilisé pour parcourir l'expression

Principe :
L'algorithme consiste à la création d'un dictionnaire représenté par un arbre à partir d'une expression algébrique. L'expression étant considérée comme sans erreur nous n'avons pas besoin de faire les vérifications la concernant.
On répète les actions suivantes tant qu'on n'a pas atteint la fin de l'expression.

On distinguera 5 cas correspondant aux 5 caractères possibles :

- « (« : A la prochaine itération nous devons empiler donc on met le booléen stack à vrai
- «) » : On doit revenir au parent de tous les nœuds à l'intérieur de cette parenthèse. Par conséquent, on dépile et on range la valeur dans le pointeur courant.
- « + » : Ce caractère signifie que le prochain caractère sera ajouté par lien horizontal. On dépile dans le pointeur courant pour revenir au père et on fait pointer prec sur l'adresse du lien horizontal du pointeur courant. Finalement on, passe stack à vrai pour empiler la prochaine valeur.
- « * » : On passe stack à faux. Il ne faut pas empiler le prochain caractère. De plus, prec pointe sur l'adresse du lien vertical du pointeur courant.
- « A-Z || a-z » : On créé un nouveau nœud. Si stack est à vrai, on empile pointeur courant et on passe stack a faux. Finalement on ajoute le nouveau nœud dans l'arbre.

A la fin de cette fonction on libère la pile.

AFFICHAGE

Affichage des mots du dictionnaire dans l'ordre alphabétique.
Lexique :
tree : l'arbre représentant le dictionnaire à afficher
word : le mot à afficher
i : un entier pour parcourir le mot word
ptrCour : Pointeur courant sur l'arbre
pile : pile utiliser pour revenir à certains nœuds

Principe : L'algorithme consiste au parcours de l'arbre tant qu'on n'arrive pas à la fin. C'est donc notre boucle principale.
Boucle principale : On regarde si la lettre courante est en majuscule ou non, car une majuscule indique la fin d'un mot. Si c'est une majuscule on affiche le mot. Dans le cas contraire on ajoute la lettre a word (le mot n'est pas finis).
Ensuite on empile le pointeur courant. On sauvegarde la position dans le cas où ce nœud à des frères. Le pointeur courant continue en suivant le lien vertical.
Si on arrive à un pointeur courant vide ou dépile pour remonter dans l'arbre jusqu'à trouver un autre chemin pour faire des mots.
On répète cela jusqu'à afficher tous les mots du dictionnaire.

INSERTION

Insertion d'un mot dans le dictionnaire. Le traitement est sans effet si le mot est déjà présent dans le dictionnaire. Le traitement doit prendre en compte l'insertion dans un arbre vide.

Lexique :
Tree : arbre dans lequel on veut
Word : le mot à insérer
Prec : double pointeur sur l'arbre tree

Cour : pointeur courant sur l'arbre
Elt : pointeur sur un nœud à insérer
i : entier utiliser pour parcourir le mot à insérer
continuer : booléen indiquant si on continue à insérer ou non

Principe : L'algorithme consiste à insérer lettre par lettre le mot à insérer dans le dictionnaire. Donc tant qu'il nous reste des lettres à insérer on continue. Ensuite, on recherche si le caractère à insérer est présent dans l'arbre.

On distinguera deux cas :

- Le caractère est déjà présent dans l'arbre => Le pointeur courant pointe sur le lien vertical du pointeur prec. Afin d'insérer le prochain caractère au bon endroit s'il n'est pas déjà présent.
- Le caractère n'est pas présent dans l'arbre donc on insère un nouveau nœud correspondant à la lettre courante à insérer.

On insère le dernier caractère en majuscule pour indiquer la fin du mot.

RECHERCHE

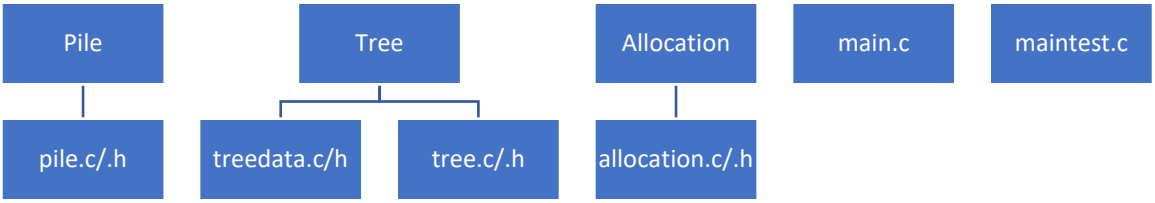
Recherche dans le dictionnaire de tous les mots qui commencent par un motif donné. L'algorithme affiche donc tous les mots commençant par le motif donné en entrer.

Lexique :
Tree : arbre dans lequel rechercher les mots
Motif : le motif à chercher
Prec : double pointeur sur l'arbre
i : entier
continue : booléen

Principe : On recherche dans l'arbre le motif et on récupère le pointeur sur son emplacement. Puis on affiche l'arbre à partir de ce nœud donc tous les mots commençant par le motif en entrer.

Organisation du code source :

ARBORESCENCE



Pile : Gestion de la pile

Tree : Gestion de l'arbre

Allocation : Gestion des allocations mémoires des pointeurs

main : Fichier principal

maintest : Fichier de test pour les cas particuliers énoncés dans le rapport

MAKEFILE

Le fichier makefile permet la compilation du fichier main.c et test.c en deux fichiers exécutables.

```
CC=gcc
OPT= -g -Wall -Wextra

all : prog test
prog : main.o tree.o specific_tree.o pile.o allocation.o
    $(CC) -o prog main.o tree.o specific_tree.o pile.o allocation.o
test : maintest.o tree.o specific_tree.o pile.o allocation.o
    $(CC) -o test maintest.o tree.o specific_tree.o pile.o allocation.o
    rm *.o
main.o : main.c Tree/tree.h
    $(CC) -c main.c $(OPT)
maintest.o : maintest.c Tree/tree.h
    $(CC) -c maintest.c $(OPT)
tree.o : Tree/tree.c Tree/tree.h Tree/specific_tree.h
    $(CC) -c Tree/tree.c $(OPT)
specific_tree.o : Tree/specific_tree.c Tree/specific_tree.h
    $(CC) -c Tree/specific_tree.c $(OPT)
pile.o : Pile/pile.c Pile/pile.h
    $(CC) -c Pile/pile.c $(OPT)
allocation.o : Allocation/allocation.c Allocation/allocation.h
    $(CC) -c Allocation/allocation.c $(OPT)
```

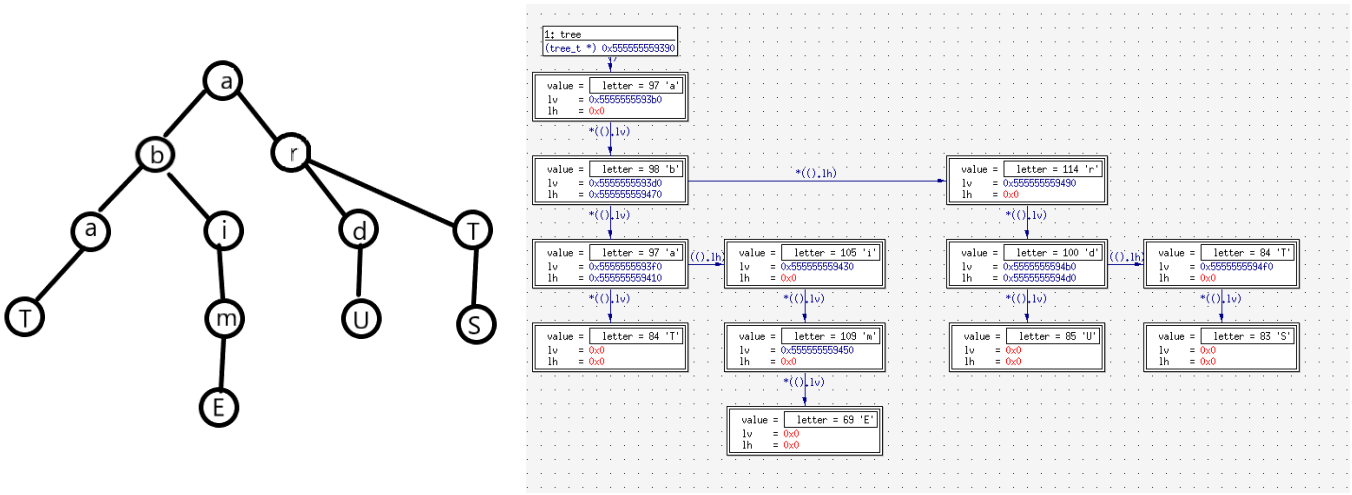
Jaune : Le fichier source est traité par le préprocesseur. Il remplace les chaînes de caractères, inclus les fichiers sources…). Puis, ce fichier est traduit par le préprocesseur en assembleur (suite d'instruction associées aux fonctionnalités du processeur). Finalement, le fichier est transformé en code assembleur compréhensible par le processeur.
Vert : Les fichiers nécessaires sont liés entre eux afin de produire un fichier exécutable (ici prog et test)
Rouge : permet de produire deux fichiers exécutables.

Vérification Mémoire :

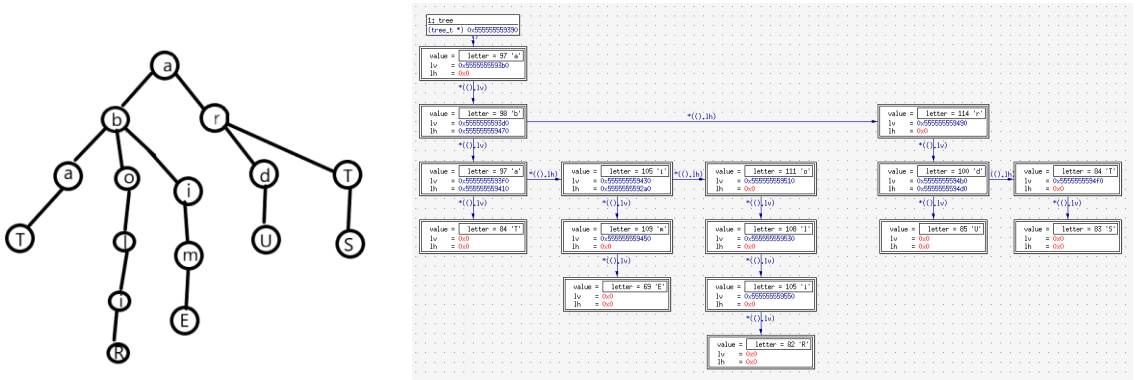
On utilise DDD afin de contrôler la bonne création de l'arbre en utilisant la fonction « tree_t * createTree(char* expression) » qui consiste à la création d'un dictionnaire sous forme d'arbre à partir d'une expression algébrique.

Ici, nous utilisons l'expression algébrique suivante : "(a*(b*(a*T+i*m*E)+r*(d*U+T*S)))"

Arbre attendu à gauche et DDD à droite, Arbre avant insertion d'un nouveau mot :



Arbre attendu à gauche et DDD à droite, Arbre après insertion d'un nouveau mot : « Abolir »



Grâce à DDD on peut constater que la création d'un arbre fonctionne correctement. L'arbre visualisé avec DDD correspond à celui attendu que ce soit avant et après l'insertion d'un mot.

Valgrind : capture d'écran après compilation et exécution du fichier « main.c » :

Afin de lancer le main, il faut :
-taper make dans le terminal à la racine du projet
-Exécuter l'exécutable « prog »

```
==53476==
==53476== HEAP SUMMARY:
==53476==     in use at exit: 0 bytes in 0 blocks
==53476==   total heap usage: 23 allocs, 23 frees, 2,056 bytes allocated
==53476==
==53476== All heap blocks were freed -- no leaks are possible
==53476==
==53476== For lists of detected and suppressed errors, rerun with: -s
==53476== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

On remarque qu'il n'y a aucune fuite de mémoire. Toutes les zones mémoires allouées sont libérées à la fin du programme.

Jeux de test :

Afin de lancer les tests, il faut :
-taper make dans le terminal à la racine du projet
-Exécuter l'exécutable « test »

Les captures d'écrans correspondent aux algorithmes associés. Le code rendu exécute l'entièreté des tests en même temps.

Test des fonctions de « tree.h » :

```
tree_t * createTree(char* expression);
```

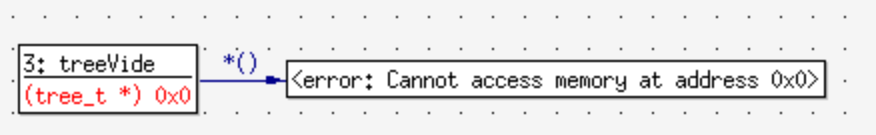
Liste des cas :

- -Arbre Vide
- -Arbre Non Vide :
  - -Plusieurs arbres distincts
  - -Un seul arbre

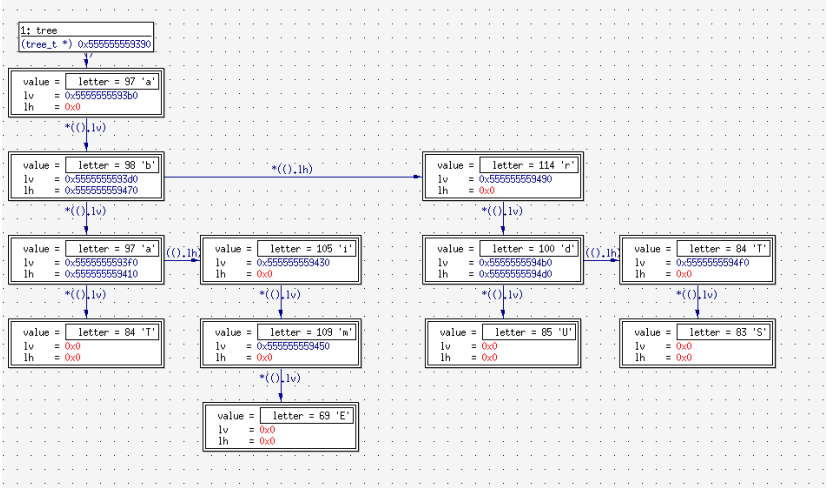
Résultat de l'exécution :

```
Création d'un arbre vide

Libération de l'arbre

Création d'un arbre rempli

Libération de l'arbre

==310==
==310== HEAP SUMMARY:
==310==     in use at exit: 0 bytes in 0 blocks
==310==   total heap usage: 17 allocs, 17 frees, 4,816 bytes allocated
==310==
==310== All heap blocks were freed -- no leaks are possible
==310==
==310== For lists of detected and suppressed errors, rerun with: -s
==310== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Arbre Vide :



Arbre Non Vide :



Explication :

Dans le premier cas on teste la création d'un arbre vide, donc en entrant en paramètre une expression vide (« »).
Dans le second cas nous entrons une expression non vide "(a*(b*(a*T+i*m*E)+r*(d*U+T*S)))".
Dans les deux cas on observe via DDD que les arbres sont correctement créés. De plus Valgrind nous indique qu'aucune fuite de mémoire est à déplorer.

```
void displayDictionary(tree_t ** tree, char * word, int i);
```

Liste des cas :

-word est vide => Il faut tout afficher

-word est plein

=>Afficher tous les mots qui ont le même paterne

=>Aucun mot du même paterne on affiche un message d'erreur

Résultat de l'exécution :

```
Création d'un arbre vide

Affichage de l'arbre vide

Libération de l'arbre

Création d'un arbre rempli

Affichage de l'arbre rempli
abat
abime
ardu
art
arts

Libération de l'arbre

==338==
==338== HEAP SUMMARY:
==338==     in use at exit: 0 bytes in 0 blocks
==338==   total heap usage: 21 allocs, 21 frees, 5,248 bytes allocated
==338==
==338== All heap blocks were freed -- no leaks are possible
==338==
==338== For lists of detected and suppressed errors, rerun with: -s
```

Explication :

Dans le cas d'un arbre vide, la fonction d'affichage des mots d'un dictionnaire nous retourne bien aucun mot. Dans le deuxième cas, il nous retourne bien les mots attendus en considérant que l'expression utilisé est la suivante : "(a*(b*(a*T+i*m*E)+r*(d*U+T*S)))"

```
void insert_word(tree_t ** tree, char * word);
```

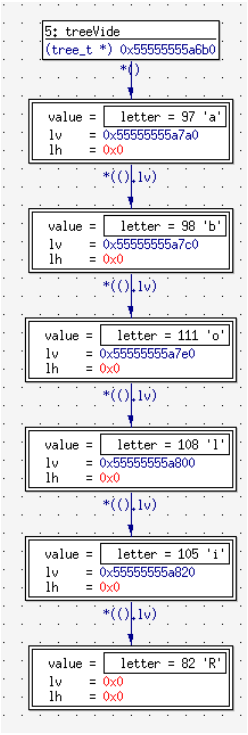
Liste des cas :

- -Arbre Non Vide
  - -Insérer mot non existant
  - -Insérer mot existant
- -Arbre Vide
  - -Insertion mot

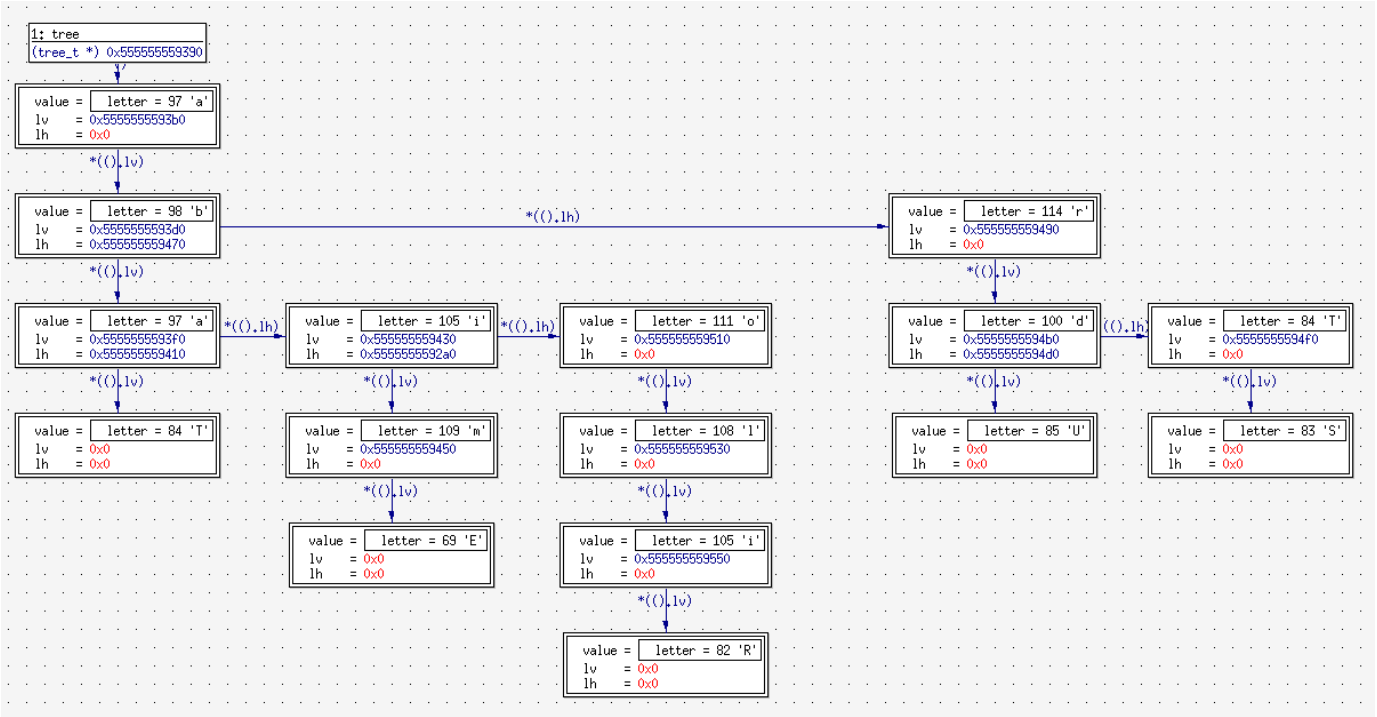
Résultat de l'exécution :

```
Création d'un arbre vide

Affichage de l'arbre vide

Insertion d'un mot dans l'arbre vide

Affichage de l'arbre avec le nouveau mot
abolir

Libération de l'arbre

Création d'un arbre rempli

Affichage de l'arbre rempli
abat
abime
ardu
art
arts

Insertion d'un mot dans l'arbre rempli

Affichage de l'arbre avec le nouveau mot
abat
abime
abolir
ardu
art
arts

Libération de l'arbre

==366==
==366== HEAP SUMMARY:
==366==     in use at exit: 0 bytes in 0 blocks
==366==   total heap usage: 35 allocs, 35 frees, 5,920 bytes allocated
==366==
==366== All heap blocks were freed -- no leaks are possible
==366==
==366== For lists of detected and suppressed errors, rerun with: -s
==366== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Arbre Vide après insertion :



Arbre non Vide après Insertion :



Explication :

Dans le cas de l'arbre vide, on remarque que le mot « abolir » a bien été insérer dans l'arbre, l'arbre étant vide cela à juste créé un arbre avec un mot. (On compare à la capture du test de création d'arbre).

Dans le cas de l'arbre non vide, le mot « abolir » est bien insérer dans l'arbre et à la bonne place, c'est-à-dire dans l'ordre alphabétique. De même, on peut comparé avec la capture du teste de création d'arbre.

```c
void search_pattern(tree_t ** tree, char * motif);
```

Liste des cas :

-Un ou plusieurs mots correspondant au motif

-Aucune mot ne correspond au motif

Résultat de l'exécution :

```
Création d'un arbre vide

Affichage de l'arbre vide

Recherche du paterne 'ab' dans l'arbre vide
Aucun mot correspondant au motif
Insertion d'un mot dans l'arbre vide

Affichage de l'arbre avec le nouveau mot
abolir
```

```
Recherche du paterne 'ab' dans l'arbre avec le nouveau mot
abolir

Libération de l'arbre

Création d'un arbre rempli

Affichage de l'arbre rempli
abat
abime
ardu
art
arts

Recherche du paterne 'ab' dans l'arbre rempli
abat
abime

Insertion d'un mot dans l'arbre rempli

Affichage de l'arbre avec le nouveau mot
abat
abime
abolir
ardu
art
arts

Recherche du patterne 'ab' dans l'arbre vide
abat
abime
abolir

Libération de l'arbre

==397==
==397== HEAP SUMMARY:
==397==     in use at exit: 0 bytes in 0 blocks
==397==   total heap usage: 41 allocs, 41 frees, 6,568 bytes allocated
==397==
==397== All heap blocks were freed -- no leaks are possible
==397==
==397== For lists of detected and suppressed errors, rerun with: -s
==397== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Explication :

On retrouve bien les mots correspondant au motif entrer en paramètre.