Arbres binaire de recherche - Visualisation.

Structures de données - Travaux dirigés sur machines

Séance 6 et 7

Il est demandé aux étudiants de réaliser chaque exercice dans un répertoire séparé.
Les travaux seront réalisés à partir de l'archive `SD-TP6.tgz` fournie sur moodle et contenant la hiérarchie suivante :

`SD-TP6`
–> `Test` : répertoire contenant les fichiers de test.
–> `Code` : répertoire contenant le code source fourni devant être complété.

Il est demandé aux étudiants de rédiger, dans un fichier texte contenu dans le dossier correspondant, les réponses aux questions posées pour chaque exercice.
À la demande de l'enseignant, l'étudiant devra pouvoir fournir une archive similaire à l'archive de départ contenant le résultat de son travail.

Table des matières

L'objectif de ce TP, qui court sur 2 séances, est de programmer le module de gestion des arbres binaires de recherche et d'en effectuer la visualisation en utilisant l'outil `dot` de la suite logicielle graphviz [1] comme montré sur la figure 1.
Le planning à suivre sur ces deux séances est le suivant :

Séance 1 Réalisation des exercices 1 à 4 inclus.
Séance 2 Réalisation des exercices 5 à 7 inclus.

1. https://www.graphviz.org/

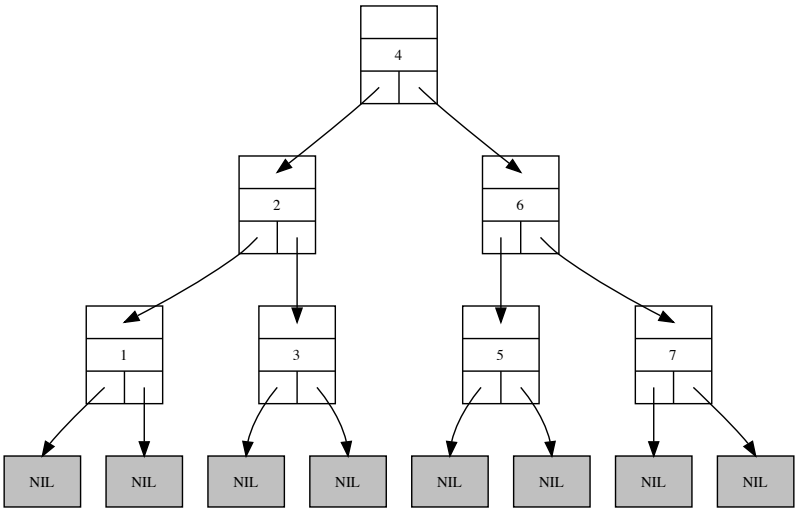

FIGURE 1 – Visualisation d'un arbre binaire de recherche équilibré

1 Description de l'archive logicielle fournie

L'archive logicielle fournie pour ce TP comprend :

- Un module de gestion de file, `queue.h` et `queue.c`, implantant le TAD QUEUE vu en cours et déjà utilisé dans les TPs précédent.
- Un module de gestion d'arbre binaire de recherche, `bstree.h` et `bstree.c`, proposant une implantation des arbres binaires généraux telle que vue en cours et définissant l'interface des opérations de dictionnaires et des fonctions spécifiques aux arbres binaire de recherche. Ces fonctions seront à développer dans les exercices de ce TP.
- Un programme principal, `main.c`, effectuant les tests des différents exercices. Ce programme ne devra pas être modifié, seule la définition de constantes correspondant aux exercices effectués devront être décommentées.
- Un fichier `Makefile` permettant de compiler et de générer les fichiers pdf de visualisation des arbres.
- La documentation de l'archive fournie, à générer par la commande `make doc`.

Les développements à effectuer pendant ce TP ne concernent donc que le fichier `bstree.c` et l'implantation des fonctions indiquées dans le sujet. **Attention**, le résultat de ce TP servira de base de développement pour les TPs suivants. Une attention particulière devra donc être apportée sur la réutilisabilité et l'extensibilité du module développé.

2 Travail à réaliser

Le but de ce TP est d'implanter les opérations de dictionnaires en utilisant le TAD *Arbre binaire de recherche* vu en cours.

Tous les développements demandés doivent être réalisés dans le fichier `bstree.c` dans lequel les opérateurs à programmer sont définis mais vides ou réduits à des instructions permettant la compilation et qu'il faudra donc remplacer. Pendant ce TP, aucune modification d'interface n'est nécessaire.

Un arbre binaire de recherche est un arbre binaire respectant l'invariant de structure suivant, pour tous les nœuds de l'arbre :

```
root(left(t)) < root(t) < root(right(t))
```

Si un fils est manquant, on considère que la propriété est vérifiée concernant ce fils.

Sorte : BinarySearchTree
Utilise : int, bool

Opérateurs_Constructeurs :

bstree_create : → BinarySearchTree
bstree_cons : BinarySearchTree × BinarySearchTree × int → BinarySearchTree

Opérateurs :

bstree_empty : BinarySearchTree → bool
bstree_root : BinarySearchTree → int
bstree_left : BinarySearchTree → BinarySearchTree
bstree_right : BinarySearchTree → BinarySearchTree
bstree_add : BinarySearchTree × int → BinarySearchTree
bstree_search : BinarySearchTree × int → bool
bstree_remove : BinarySearchTree × int → BinarySearchTree

Préconditions :

bstree_cons(l, r, k) ***défini ssi*** *bstree_empty(l)* ∧ *bstree_empty(r)*
bstree_root(t), bstree_left(t), bstree_right(t) ***défini ssi*** ¬*bstree_empty(t)*

Axiomes :

$bstree\_empty(bstree\_create) = true$
$bstree\_empty(bstree\_cons(l, r, v)) = false$
$bstree\_root(bstree\_cons(l, r, v)) = v$
$bstree\_left(bstree\_cons(l, r, v)) = l$
$bstree\_right(bstree\_cons(l, r, v)) = r$

$bstree\_add(bstree\_create, i) = bstree\_cons(bstree\_create, bstree\_create, i)$
$i < x \rightarrow bstree\_add(bstree\_cons(l, r, x), i) = bstree\_cons(bstree\_add(l, i), r, x)$
$i = x \rightarrow bstree\_add(bstree\_cons(l, r, x), i) = bstree\_cons(l, r, x)$
$i > x \rightarrow bstree\_add(bstree\_cons(l, r, x), i) = bstree\_cons(l, bstree\_add(r, i), x)$

$bstree\_search(bstree\_create, i) = false$
$i < x \rightarrow bstree\_search(bstree\_cons(l, r, x), i) = bstree\_search(l, i)$
$i = x \rightarrow bstree\_search(bstree\_cons(l, r, x), i) = true$
$i > x \rightarrow bstree\_search(bstree\_cons(l, r, x), i) = bstree\_search(r, i)$

FIGURE 2 – Spécification des arbres binaire de recherche

L'implantation proposée dans le module `bstree.h / bstree.c` correspond à la spécification décrite en figure 2 vue en cours. Il est à noter que, pour pouvoir établir et conserver l'invariant de structure d'arbre de recherche, le constructeur `bstree_cons(l, r, k)` ne pourra être utilisé que pour construire les feuilles. Dans l'implantation proposée, ce constructeur `bstree_cons` à été

rendu privé au module, en l'enlevant de l'interface (`bstree.h`) et en le définissant uniquement dans l'implantation (`bstree.c`). Cela rend aussi possible une implantation des opérations de dictionnaire d'une façon plus efficace que la traduction directe des axiomes, en tenant compte de la représentation interne de l'arbre et en assurant la cohérence globale et la validité de la structure de données.
De même, l'opérateur `bstree_parent` a été rajouté par choix d'implantation afin de simplifier l'écriture de nombreux opérateurs.
Il est à noter que la spécification donnée ici n'est pas suffisamment complète, l'écriture des axiomes décrivant l'opérateur `bstree_remove` alant au delà de ce que nous avons vu ce semestre.
L'implantation proposée représente un arbre comme un pointeur vers sa racine. Un arbre vide, retourné par exemple par le constructeur `bstree_create` sera donc représenté par le pointeur `NULL`.
Selon la même démarche que dans les TPs précédents, le type `BinarySearchTree` est défini par `typedef struct _bstree BinarySearchTree;` et l'utilisateur ne peut donc que déclarer un arbre par `BinarySearchTree *theTree`. Le type défini par `typedef BinarySearchTree *ptrBinarySearchTree;` permet de définir simplement un pointeur vers un arbre.

2.1 Exercice 1 : implantation de l'opérateur `bstree_add`

L'implantation fournie du module `bstree` contient la définition de la représentation interne d'un arbre binaire et l'implantation des différents opérateurs communs à tous les arbres binaire.
Dans ce premier exercice, il est demandé d'implanter, **de façon itérative**, la fonction `void bstree_add(ptrBinarySearchTree *t, int v)` dont la définition est vide dans le fichier `bstree.c`.
Cette fonction reçoit en paramètre l'adresse d'un pointeur vers un arbre binaire de recherche, éventuellement vide, (`ptrBinarySearchTree *t`) et la valeur à ajouter dans l'arbre.
On rappelle que dans les arbres binaires de recherche, l'ajout de nouvelle valeur se fait sur les feuilles.
Le principe d'ajout d'une valeur dans l'arbre est le suivant :

- En partant de la racine de l'arbre, l'algorithme parcourt la branche au bout de laquelle doit être ajoutée la valeur. Ce parcours mémorise l'adresse où doit être ajoutée la feuille (`ptrBinarySearchTree *cur`, initialisée à `t`) et le nœud qui sera le parent de la feuille ajoutée (`BinarySearchTree *par`, initialement à `NULL`) correspondant à l'ajout à la racine de l'arbre.
- En fin de parcours, une nouvelle feuille est crée en utilisant l'opérateur `BinarySearchTree *bstree_cons(BinarySearchTree *left, BinarySearchTree *right, int root)` et la relation vers le parent est mise à jour sur la nouvelle feuille.

Après avoir programmé cet opérateur, il faut alors décommenter la ligne *`//#define EXERCICE_1`* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Le résultat attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
```

2.2 Exercice 2 : parcours en profondeur d'abord de l'arbre.

Comme toute collection, un arbre binaire de recherche doit pouvoir être parcouru pour effectuer un traitement sur chaque nœud de l'arbre. Toutefois, ce traitement ne doit pas remettre en cause

l'invariant de structure des arbres binaires de recherche. Nous définissons donc le traitement à appliquer sur un nœud de l'arbre par une fonction du type `typedef void(*OperateFunctor)(const BinarySearchTree *, void *);`.
Une telle fonction reçoit comme premier paramètre le nœud devant être traité (la racine d'un sous-arbre) et un pointeur non typé vers un éventuel paramètre fourni par l'utilisateur pour réaliser son traitement. Le nœud étant de type `const BinarySearchTree *`, il ne pourra pas être modifié par le foncteur sous peine d'erreur de compilation.
Écrire, **de façon récursive**, les 3 fonctions suivantes permettant d'effectuer un parcours en profondeur d'abord de façon préfixe, infixe et postfixe.

1. `void bstree_depth_prefix(const BinarySearchTree *t, OperateFunctor f, void *userData);`
2. `void bstree_depth_infix(const BinarySearchTree *t, OperateFunctor f, void *userData);`
3. `void bstree_depth_postfix(const BinarySearchTree *t, OperateFunctor f, void *userData);`

Après avoir programmé ces opérateurs, il faut alors décommenter la ligne *`//#define EXERCICE_2`* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Le résultat attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Visiting the tree.
        Prefix visitor = 4 2 1 3 6 5 7
        Infix visitor = 1 2 3 4 5 6 7
        Postfix visitor = 1 3 2 5 7 6 4
Done.
```

Que remarquez-vous sur la sortie de vos parcours. ?

2.3 Exercice 3 : parcours en largeur d'abord de l'arbre.

Il peut être utile, pour effectuer certains traitements de parcourir l'arbre niveau par niveau. Bien que ne tirant pas profit de l'ordre induit par l'invariant de structure des arbres binaires de recherche, ce type de parcours, en largeur d'abord, sertdans notre exemple à effectuer une visualisation de l'arbre.

En utilisant l'implantation fournie du TAD Queue (fichiers `queue.h / queue.c`) et sans modifier ces fichiers, programmez la fonction
`void bstree_iterative_breadth_prefix(const BinarySearchTree *t, OperateFunctor f, void *userData);`
effectuant un parcours en largeur d'abord de l'arbre et qui applique le foncteur `f` sur tous les nœuds de l'arbre en transmettant le paramètre `userData`.
Ce parcours est utilisé dans le programme principal pour générer un fichier de description de l'arbre en langage ***dot*** pour le visualiser. Dans le fichier `main.c`, la fonction
`void node_to_dot(const BinarySearchTree *t, void *userData)`
permet de traduire chaque nœud en l'ensemble de commandes ***dot*** nécessaire à son dessin et affiche aussi, sur le flux standard de sortie la valeur du nœud.

À partir des données du fichier `Test/testfilesimple.txt`, le fichier dot `FullTree.dot` généré, programmant l'arbre de la figure 1, contient la description suivante :

```
digraph BinarySearchTree {
        graph [ranksep=0.5];
        node [shape = record];
        n4 [label="{{<parent>}|4|{<left>|<right>}}"];
        n4:left:c -> n2:parent:c [headclip=false, tailclip=false]
        n4:right:c -> n6:parent:c [headclip=false, tailclip=false]
        n2 [label="{{<parent>}|2|{<left>|<right>}}"];
        n2:left:c -> n1:parent:c [headclip=false, tailclip=false]
        n2:right:c -> n3:parent:c [headclip=false, tailclip=false]
        n6 [label="{{<parent>}|6|{<left>|<right>}}"];
        n6:left:c -> n5:parent:c [headclip=false, tailclip=false]
        n6:right:c -> n7:parent:c [headclip=false, tailclip=false]
        n1 [label="{{<parent>}|1|{<left>|<right>}}"];
        lnil1 [style=filled, fillcolor=grey, label="NIL"];
        n1:left:c -> lnil1:n [headclip=false, tailclip=false]
        rnil1 [style=filled, fillcolor=grey, label="NIL"];
        n1:right:c -> rnil1:n [headclip=false, tailclip=false]
        n3 [label="{{<parent>}|3|{<left>|<right>}}"];
        lnil3 [style=filled, fillcolor=grey, label="NIL"];
        n3:left:c -> lnil3:n [headclip=false, tailclip=false]
        rnil3 [style=filled, fillcolor=grey, label="NIL"];
        n3:right:c -> rnil3:n [headclip=false, tailclip=false]
        n5 [label="{{<parent>}|5|{<left>|<right>}}"];
        lnil5 [style=filled, fillcolor=grey, label="NIL"];
        n5:left:c -> lnil5:n [headclip=false, tailclip=false]
        rnil5 [style=filled, fillcolor=grey, label="NIL"];
        n5:right:c -> rnil5:n [headclip=false, tailclip=false]
        n7 [label="{{<parent>}|7|{<left>|<right>}}"];
        lnil7 [style=filled, fillcolor=grey, label="NIL"];
        n7:left:c -> lnil7:n [headclip=false, tailclip=false]
        rnil7 [style=filled, fillcolor=grey, label="NIL"];
        n7:right:c -> rnil7:n [headclip=false, tailclip=false]
}
```

Après avoir programmé cet opérateur, il faut alors décommenter la ligne *//#define EXERCICE_3* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Vous pouvez vérifier la bonne génération du fichier `FullTree.dot` en utilisant la commande `dot -Tpdf FullTree.dot -O` pour générer un fichier pdf de même nom contenant l'arbre de la figure 1.
Lors de l'exécution du programme, l'affichage attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Visiting the tree.
        Prefix visitor = 4 2 1 3 6 5 7
        Infix visitor = 1 2 3 4 5 6 7
        Postfix visitor = 1 3 2 5 7 6 4
Done.
Exporting the tree.
        4 2 6 1 3 5 7
Done.
```

Que remarquez-vous sur la sortie de votre nouveau parcours ?

2.4 Exercice 4 : recherche d'une valeur dans l'arbre.

Programmer, **de manière itérative**, la fonction
`bool bstree_search(const BinarySearchTree *t, int v);`
qui renvoie vrai si la valeur `v` se trouve dans l'arbre `t` et faux sinon.
Après avoir programmé cet opérateur, il faut alors décommenter la ligne *`//#define EXERCICE_4`* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Le résultat attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Visiting the tree.
        Prefix visitor = 4 2 1 3 6 5 7
        Infix visitor = 1 2 3 4 5 6 7
        Postfix visitor = 1 3 2 5 7 6 4
Done.
Exporting the tree.
        4 2 6 1 3 5 7
Done.
Searching into the tree.
        Searching for value 2 in the tree : true
        Searching for value 5 in the tree : true
        Searching for value 8 in the tree : false
        Searching for value 1 in the tree : true
Done.
```

2.5 Exercice 5 : suppression d'une valeur de l'arbre.

Si l'insertion d'un nœud dans un arbre binaire de recherche est une opération simple se comportant toujours de la même manière (l'arbre grandit par les feuilles), la suppression d'un nœud peut se faire différemment en fonction du nœud à supprimer.
Plusieurs cas sont à considérer, une fois que le nœud à supprimer a été trouvé à partir de sa clé :

1. **Suppression d'une feuille** :
   Il suffit de l'enlever de l'arbre vu qu'elle n'a pas de fils. La feuille est détruite et le lien qui pointait vers cette feuille dans le nœud père est mis à jour.
2. **Suppression d'un nœud n'ayant qu'un fils** :
   Il faut l'enlever de l'arbre en le remplaçant par son fils. Le nœud est détruit et le lien qui pointait vers ce nœud dans le nœud père est mis à jour pour désigner le fils du nœud supprimé.
3. **Suppression d'un nœud avec deux fils** :
   Supposons que le nœud à supprimer soit appelé N (le nœud de valeur 7 dans la figure 3). On échange le nœud N avec son successeur le plus proche (le nœud le plus à gauche du sous-arbre droit - ci-dessous, le nœud de valeur 9) ou son plus proche prédécesseur (le nœud le plus à droite du sous-arbre gauche - ci-dessous, le nœud de valeur 6). Ce

remplacement conserve la propriété d'ordonnancement local des nœuds dans un arbre binaire de recherche.

De plus, si le nœud N a pris la place de son successeur (respectivement prédécesseur), il n'a pas de fils gauche (respectivement pas de fils droit). Il est alors possible d'appliquer le cas 2 ci-dessus pour supprimer ce nœud de l'arbre.

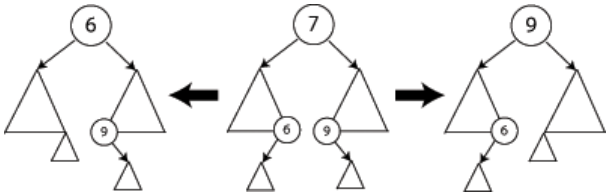

FIGURE 3 – Suppression d'un nœud dans un arbre binaire de recherche

1. Programmer la fonction

   `BinarySearchTree *bstree_successor(const BinarySearchTree *x)`

   qui renvoie le nœud successeur du nœud `x` dans l'arbre.

2. Programmer la fonction

   `BinarySearchTree *bstree_predecessor(const BinarySearchTree *x)`

   qui renvoie le nœud prédécesseur du nœud `x` dans l'arbre.

3. Programmer la fonction

   `void bstree_swap_nodes(ptrBinarySearchTree *tree, ptrBinarySearchTree from, ptrBinarySearchTre to);` qui échange, dans l'arbre `tree` les nœuds `from` et `to`.

4. En suivant l'algorithme décrit ci-dessus, programmer la fonction

   `void bstree_remove_node(ptrBinarySearchTree *t, ptrBinarySearchTree current)` qui supprime, dans l'arbre `t` le nœud `current`.

   Pour cette fonction si le nœud à supprimer possède deux fils, il sera échangé dans l'arbre avec son successeur avant d'être supprimé.

5. Programmer la fonction

   `void bstree_remove(ptrBinarySearchTree *t, int v);` qui recherche le nœud contenant la clé `v` et le supprime de l'arbre s'il existe.

6. Afin de pouvoir générer la visualisation de l'arbre résultant par une autre méthode que dans l'exercice 3, programmer la fonction

   `void bstree_iterative_depth_infix(const BinarySearchTree *t, OperateFunctor f, void *userData);`

   qui effectue un parcours infixe de l'arbre en profondeur d'abord de façon itérative, sans utilisation de structure de donnée annexe et sans modification de l'arbre.

Après avoir programmé ces opérateurs, il faut alors décommenter la ligne *`//#define EXERCICE_5`* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Le résultat attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Visiting the tree.
```

```
        Prefix visitor = 4 2 1 3 6 5 7
        Infix visitor = 1 2 3 4 5 6 7
        Postfix visitor = 1 3 2 5 7 6 4
Done.
Exporting the tree.
        4 2 6 1 3 5 7
Done.
Searching into the tree.
        Searching for value 2 in the tree : true
        Searching for value 5 in the tree : true
        Searching for value 8 in the tree : false
        Searching for value 1 in the tree : true
Done.
Removing from the tree.
        Removing the value 2 from the tree :    1 3 4 5 6 7
        Removing the value 5 from the tree :    1 3 4 6 7
        Removing the value 8 from the tree :    1 3 4 6 7
        Removing the value 3 from the tree :    1 4 6 7
        Removing the value 1 from the tree :    4 6 7
Done.
```

Vous pouvez générer les fichiers pdf de visualisation de l'arbre à partir des fichiers dot produits par le programme principal en exécutant `make pdf`.
Que constatez-vous si vous remplacez un nœud ayant deux fils lors de sa suppression par son prédécesseur au lieu de son successeur ?
Expliquez pourquoi, alors que le même foncteur que dans l'exercice 3 est utilisé dans le programme principal pour générer le fichier dot et afficher les nœuds dans l'ordre de leur visite, l'affichage n'est pas le même que l'affichage de l'exercice 3.

2.6 Exercice 6 : itérateurs sur un arbre.

Comme toute collection ordonnée, un arbre binaire de recherche doit pouvoir être parcouru dans l'ordre de ses clés. Si le visiteur `bstree_depth_infix` permet en effet de parcourir l'arbre dans l'ordre croissant de ses clés, il est cependant limité sur deux points :

- Seul un traitement représenté par un foncteur de type `OperateFunctor` peut être appliqué sur le nœud. Il n'est donc pas possible simplement d'exploiter la valeur d'un nœud en fonction des autres valeurs de l'arbre.
- Le parcours de l'arbre se fait toujours dans l'ordre croissant des clés. Il peut parfois être utile de parcourir l'arbre dans l'ordre décroissant de ses clés.

Le patron de conception comportemental `Iterator`, déjà utilisé sur les listes, est alors utile pour pouvoir appliquer un traitement quelconque sur les valeurs des nœuds de l'arbre. Par exemple, afficher les valeurs de l'arbre dans l'ordre décroissant de leur clé comme proposé dans le programme principal :

```
    BSTreeIterator *i = bstree_iterator_create(theTree, backward);
    for (  i = bstree_iterator_begin(i);
           !bstree_iterator_end(i);
           i = bstree_iterator_next(i)
        )
        printf("%d ", bstree_root(bstree_iterator_value(i)));
    bstree_iterator_delete(&i);
```

Afin d'implanter de patron de conception pour les arbres bianire de recherche, en respectant l'interface décrite dans le fichier `bstree.h`, programmer les opérateurs suivants :

1. `const BinarySearchTree *goto_min(const BinarySearchTree *e)` qui renvoie la feuille de clé minimale dans l'arbre `e`
2. `const BinarySearchTree *goto_max(const BinarySearchTree *e)` qui renvoie la feuille de clé maximale dans l'arbre `e`

En utilisant les opérateurs précédants ainsi que les opérateurs `bstree_successor` et `bstree_predecessor`, programmer le constructeur
`BSTreeIterator *bstree_iterator_create(const BinarySearchTree *collection, IteratorDirection direction)`
permettant de construire un itérateur sur l'arbre `collection` allant dans la direction `direction`.
Les valeurs possibles pour ce dernier paramètre sont définies dans le type énuméré `IteratorDirection`.
Après avoir programmé ces opérateurs, il faut alors décommenter la ligne *`//#define EXERCICE_6`* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en exécutant `./bstreetest ../Test/testfilesimple.txt`.
Le résultat attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Visiting the tree.
        Prefix visitor = 4 2 1 3 6 5 7
        Infix visitor = 1 2 3 4 5 6 7
        Postfix visitor = 1 3 2 5 7 6 4
Done.
Exporting the tree.
        4 2 6 1 3 5 7
Done.
Searching into the tree.
        Searching for value 2 in the tree : true
        Searching for value 5 in the tree : true
        Searching for value 8 in the tree : false
        Searching for value 1 in the tree : true
Done.
Removing from the tree.
        Removing the value 2 from the tree :    1 3 4 5 6 7
        Removing the value 5 from the tree :    1 3 4 6 7
        Removing the value 8 from the tree :    1 3 4 6 7
        Removing the value 3 from the tree :    1 4 6 7
        Removing the value 1 from the tree :    4 6 7
Done.
Iterate descending onto the tree.
        values : 7 6 4
Done.
```

2.7 Exercice 7 : destruction de l'arbre et libération des ressources.

Programmer l'opérateur `void bstree_delete(ptrBinarySearchTree *t);` libérant l'ensemble des ressources mémoire utilisées pour la gestion de l'arbre binaire de recherche `t`. Cette fonction peut être programmée de façon très simple en utilisant un des visiteurs développés précédemment, avec un foncteur approprié. Vous justifierais le choix de visiteur que vous avez réalisé. Vous veillerez à ce qu'a l'issue de cette fonction, l'arbre soit effectivement vide. Vous vérifierez avec l'outil `valgrind` que toutes les ressources mémoire allouées par le programme soient libérées.

Après avoir programmé ces opérateurs, il faut alors décommenter la ligne *//#define EXERCICE_7* dans le fichier `main.c` et vérifier la compilation (`make`) et l'exécution correcte du programme en éxécutant `./bstreetest ../Test/testfilesimple.txt`.
Le résultat attendu est le suivant :

```
$ ./bstreetest ../Test/testfilesimple.txt
Adding values to the tree.
        4 6 7 5 2 3 1
Done.
Visiting the tree.
        Prefix visitor = 4 2 1 3 6 5 7
        Infix visitor = 1 2 3 4 5 6 7
        Postfix visitor = 1 3 2 5 7 6 4
Done.
Exporting the tree.
        4 2 6 1 3 5 7
Done.
Searching into the tree.
        Searching for value 2 in the tree : true
        Searching for value 5 in the tree : true
        Searching for value 8 in the tree : false
        Searching for value 1 in the tree : true
Done.
Removing from the tree.
        Removing the value 2 from the tree :    1 3 4 5 6 7
        Removing the value 5 from the tree :    1 3 4 6 7
        Removing the value 8 from the tree :    1 3 4 6 7
        Removing the value 3 from the tree :    1 4 6 7
        Removing the value 1 from the tree :    4 6 7
Done.
Iterate descending onto the tree.
        values : 7 6 4
Done.
Deleting the tree.
Done.
```