

Arbres binaires de recherche - Visualisation.

Algorithmique 3 - Travaux pratiques

Les travaux pratiques seront réalisés à partir de l'archive `base_code_lab5.tar.gz` fournie sur moodle et contenant la hiérarchie suivante :

`base_code_lab5`

- > **Code** : répertoire contenant le code source fourni devant être complété.
- > **Test** : répertoire contenant les fichiers de test.

Bien qu'il ne soit pas exigé de dépôt pour cette séquence, il est recommandé de préparer les informations suivantes qui serviront de point de départ pour la dernière séquence du semestre. Il est demandé à l'étudiant de rédiger, dans un fichier convertible au format **pdf**, un compte rendu de travaux pratiques contenant, pour chaque exercice :

- les réponses aux éventuelles questions posées pour l'exercice dans le sujet,
- une description synthétique des algorithmes mis en œuvre et une analyse de la complexité en temps et en espace de ces algorithmes,
- une synthèse de la démarche de programmation mise en œuvre, en précisant :
 - les références concernant les documentations et aides utilisées (livres, camarades, chat-GPT, site web ...),
 - les difficultés rencontrées et comment elles ont été résolues.

L'étudiant préparera une archive selon les spécifications suivantes :

- Format de l'archive identique au format initial (.tar.gz),
- Contenu de l'archive (il n'est pas nécessaire de dupliquer les jeux de test) :
 - le compte rendu de TP au format adapté à sa modification,
 - un dossier **Code** contenant uniquement le fichier source `bstree.c`, à l'exclusion de tout autre fichier.

Cette archive devra pouvoir être transmise à l'enseignant sur demande.

Table des matières

1	Description de l'archive logicielle fournie	2
2	Travail à réaliser	3
2.1	Exercice 1 : implantation de l'opérateur <code>bstree_add</code>	4
2.2	Exercice 2 : parcours en profondeur d'abord de l'arbre.	5
2.3	Exercice 3 : parcours en largeur d'abord de l'arbre.	5
2.4	Exercice 4 : recherche d'une valeur dans l'arbre.	7
2.5	Exercice 5 : suppression d'une valeur de l'arbre.	8
2.6	Exercice 6 : itérateurs sur un arbre.	11
2.7	Exercice 7 : destruction de l'arbre et libération des ressources.	12

L'objectif de cette séquence, 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](https://www.graphviz.org/)¹ comme montré sur la figure 1.

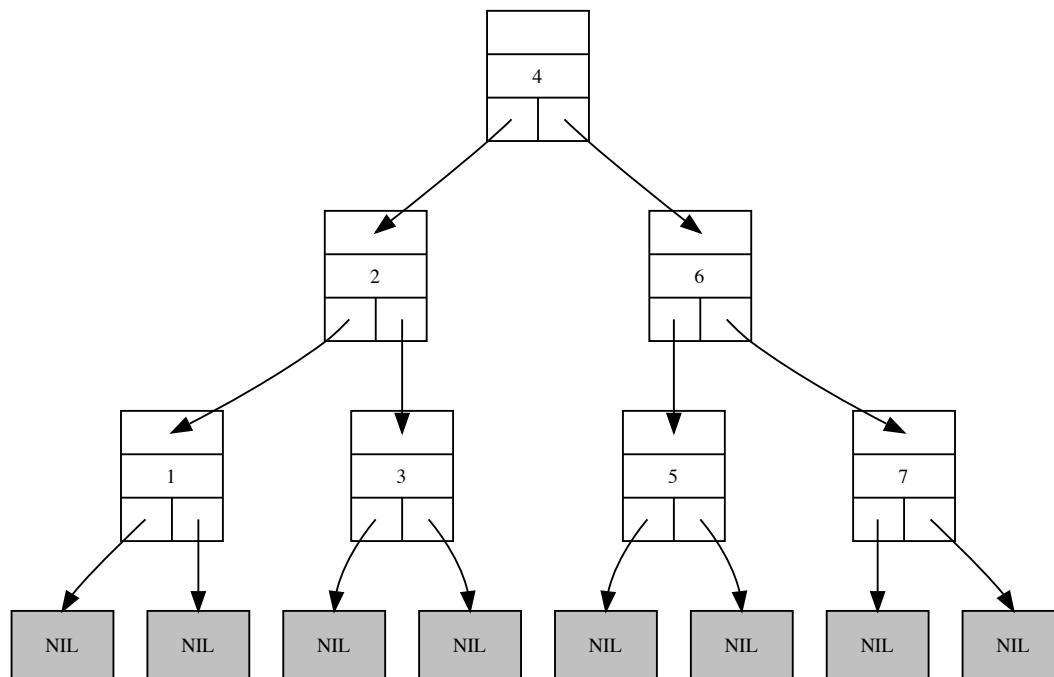


FIGURE 1 – Visualisation d'un arbre binaire de recherche équilibré

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 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 séquences précédentes.
- 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`.

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

Les développements à effectuer pendant ce TP ne concernent donc que 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.

ATTENTION, le résultat de cette séquence servira de base de développement de la séquence suivante. Une attention particulière devra donc être apportée sur la réutilisabilité et l'extensibilité du module développé.

2 Travail à réaliser

L'implantation proposée dans le module `bstree.h` / `bstree.c` correspond à la spécification décrite en figure 2 et étudiée en cours-TD.

Dans ce module, la représentation interne d'un arbre binaire et l'implantation des opérateurs communs à tous les arbres binaires sont déjà définis et seules les fonctions spécifiques aux arbres binaires de recherche devront être implantées. Pendant cette séquence, aucune modification d'interface n'est nécessaire.

Sorte : BINARYSEARCHTREE

Utilise : INT, BOOL

Opérateurs_Constructeurs :

bstree_create : \rightarrow BINARYSEARCHTREE

bstree_cons : BINARYSEARCHTREE \times BINARYSEARCHTREE \times INT \rightarrow BINARYSEARCHTREE

Opérateurs :

bstree_empty : BINARYSEARCHTREE \rightarrow BOOL

bstree_key : BINARYSEARCHTREE \rightarrow INT

bstree_left : BINARYSEARCHTREE \rightarrow BINARYSEARCHTREE

bstree_right : BINARYSEARCHTREE \rightarrow BINARYSEARCHTREE

bstree_add : BINARYSEARCHTREE \times INT \rightarrow BINARYSEARCHTREE

bstree_search : BINARYSEARCHTREE \times INT \rightarrow BINARYSEARCHTREE

bstree_remove : BINARYSEARCHTREE \times INT \rightarrow BINARYSEARCHTREE

Préconditions :

bstree_cons(*l*, *r*, *k*) **défini ssi** *bstree_empty*(*l*) \wedge *bstree_empty*(*r*)

bstree_key(*t*), *bstree_left*(*t*), *bstree_right*(*t*) **défini ssi** \neg *bstree_empty*(*t*)

Axiomes :

bstree_empty(*bstree_create*) = true

bstree_empty(*bstree_cons*(*l*, *r*, *v*)) = false

bstree_key(*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 < *bstree_key*(*t*) \rightarrow *bstree_add*(*t*, *i*) = *bstree_cons*(*bstree_add*(*bstree_left*(*t*), *i*), *bstree_right*(*t*), *x*)

i = *bstree_key*(*t*) \rightarrow *bstree_add*(*t*, *i*) = *t*

i > *bstree_key*(*t*) \rightarrow *bstree_add*(*t*, *i*) = *bstree_cons*(*bstree_left*(*t*), *bstree_add*(*bstree_right*(*t*), *i*), *x*)

bstree_search(*bstree_create*, *i*) = *bstree_create*

i < *bstree_key*(*t*) \rightarrow *bstree_search*(*t*, *i*) = *bstree_search*(*bstree_left*(*t*), *i*)

i = *bstree_key*(*t*) \rightarrow *bstree_search*(*t*, *i*) = *t*

i > *bstree_key*(*t*) \rightarrow *bstree_search*(*t*, *i*) = *bstree_search*(*bstree_right*(*t*), *i*)

FIGURE 2 – Spécification des arbres binaire de recherche

Rappel. Un arbre binaire de recherche est un arbre binaire respectant l'invariant de structure suivant : pour tous les nœuds n de l'arbre, tout nœud x du sous arbre gauche de n possède une clé plus petite que celle de n et tout nœud x du sous arbre droit de n possède une clé plus grande que celle de n . Cet invariant de structure peut se formaliser par :

$$\forall n \in t \rightarrow (\forall x \in \text{left}(n) \rightarrow \text{key}(x) < \text{key}(n) \wedge \forall x \in \text{right}(n) \rightarrow \text{key}(x) > \text{key}(n))$$

Pour pouvoir établir et conserver l'invariant de structure d'arbre de recherche, le constructeur `bstree_cons`(1, `r`, `k`) ne pourra être utilisé que pour construire les feuilles. Dans l'implantation proposée, ce constructeur `bstree_cons` a 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.

Vous remarquerez que la spécification donnée en figure 2 n'est pas suffisamment complète, l'écriture des axiomes décrivant l'opérateur `bstree_remove` allant au-delà de ce que nous avons vu ce semestre en cours-TD.

L'implantation proposée représente un arbre comme un pointeur vers sa racine. Un arbre vide, retourné par le constructeur `bstree_create` ou par lors de l'accès à un sous-arbre sur un nœud ne possédant pas le sous-arbre demandé, sera donc représenté par le pointeur `NULL`. Il en va de même pour le parent du nœud racine de l'arbre.

Selon la même démarche que dans les séquences précédentes, 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 de la clé à ajouter dans l'arbre.

On rappelle que dans les arbres binaires de recherche, l'ajout de nouvelle clé se fait sur les feuilles.

Afin de vérifier l'invariant de structure par construction (au lieu de devoir le vérifier a posteriori, ce qui serait bien plus coûteux) 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 créée 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 élément de la collection, ici chaque nœud de l'arbre. Ce traitement ne doit pas remettre en cause l'invariant de structure des arbres binaires de recherche et ne peut donc pas modifier les clés. Nous définissons donc le traitement à appliquer sur un nœud de l'arbre par une fonction du type `typedef void(*OperateFuncutor)(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 environnement 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.

Programmer, **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, OperateFuncutor f, void* environnement);`
2. `void bstree_depth_infix(const BinarySearchTree* t, OperateFuncutor f, void* environnement);`
3. `void bstree_depth_postfix(const BinarySearchTree* t, OperateFuncutor f, void* environnement);`

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, sert dans 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(const BinarySearchTree* t, OperateFuncutor f, void* environnement);` 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 `environnement`.

Dans le fichier `main.c`, la fonction `void node_to_dot(const BinarySearchTree* t, void* environment)` permet de traduire chaque nœud en commandes *dot* nécessaires à son dessin et affiche aussi, sur le flux standard de sortie la valeur du nœud.

Après avoir programmé l'opérateur `bstree_iterative_breadth`, 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`.

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>}}"];
    ln1l1 [style=filled, fillcolor=grey, label="NIL"];
    n1:left:c -> ln1l1:n [headclip=false, tailclip=false]
    rn1l1 [style=filled, fillcolor=grey, label="NIL"];
    n1:right:c -> rn1l1:n [headclip=false, tailclip=false]
    n3 [label="{{<parent>}}|3|{{<left>|<right>}}"];
    ln1l3 [style=filled, fillcolor=grey, label="NIL"];
    n3:left:c -> ln1l3:n [headclip=false, tailclip=false]
    rn1l3 [style=filled, fillcolor=grey, label="NIL"];
    n3:right:c -> rn1l3:n [headclip=false, tailclip=false]
    n5 [label="{{<parent>}}|5|{{<left>|<right>}}"];
    ln1l5 [style=filled, fillcolor=grey, label="NIL"];
    n5:left:c -> ln1l5:n [headclip=false, tailclip=false]
    rn1l5 [style=filled, fillcolor=grey, label="NIL"];
    n5:right:c -> rn1l5:n [headclip=false, tailclip=false]
    n7 [label="{{<parent>}}|7|{{<left>|<right>}}"];
    ln1l7 [style=filled, fillcolor=grey, label="NIL"];
    n7:left:c -> ln1l7:n [headclip=false, tailclip=false]
    rn1l7 [style=filled, fillcolor=grey, label="NIL"];
    n7:right:c -> rn1l7:n [headclip=false, tailclip=false]
}
```

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. Pour générer plus simplement tous les fichiers PDF issus de vos exécutions de programme, vous pouvez aussi utiliser la commande `make pdf`.

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 `const BinarySearchTree* t bstree_search(const BinarySearchTree* t, int v)` ; qui renvoie le sous-arbre de `t` dont la racine porte la clé `v`. Cette fonction renvoie un sous-arbre vide si la clé n'est pas trouvée dans `t`.
- Programmer la fonction `const BinarySearchTree* bstree_successor(const BinarySearchTree* x)` qui renvoie le nœud successeur du nœud `x` dans l'arbre. Le successeur du nœud `x` est le nœud ayant la plus petite clé parmi les nœuds ayant une clé plus grande que le nœud `x`.
- Programmer la fonction `const BinarySearchTree* bstree_predecessor(const BinarySearchTree* x)` qui renvoie le nœud prédécesseur du nœud `x` dans l'arbre. Le prédécesseur du nœud `x` est le nœud ayant la plus grande clé parmi les nœuds ayant une clé plus petite que le nœud `x`.

Après avoir programmé ces opérateurs, 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
        ... its successor is 3
        ... its predecessor is 1
    Searching for value 5 in the tree : true
        ... its successor is 6
        ... its predecessor is 4
    Searching for value 8 in the tree : false
    Searching for value 1 in the tree : true
        ... its successor is 2
        ... its predecessor does not exist
Done.

```


En analysant le code que vous avez programmé pour les opérateurs `const BinarySearchTree* bstree_successor(const BinarySearchTree* x)` et `const BinarySearchTree* bstree_predecessor(const BinarySearchTree* x)`, vous remarquerez une grande ressemblance de code, la différence venant des fonctions d'accès au fils gauche ou au fils droit qui sont symétriques dans ces deux fonctions. Afin de faciliter la maintenance et la lisibilité du code en diminuant la redondance dans le code source, il est souhaitable de remplacer cette redondance par la programmation d'un opérateur prenant en paramètres les fonctions d'accès à utiliser et à l'appeler ensuite avec les bonnes fonctions.

En identifiant les symétries permettant de passer de la recherche du successeur à la recherche du prédécesseur,

1. introduire, de façon privée au module `bstree` les types `AccessFunction` d'une fonction d'accès à un sous-arbre et `ChildAccessors` d'une structure permettant de stocker les fonctions d'accès aux sous-arbres nécessaires à la recherche d'un successeur ou d'un prédécesseur,
2. programmer la fonction privée au module, `BinarySearchTree* find_next(const BinarySearchTree* x, ChildAccessors access)`, qui effectue la recherche d'un successeur ou d'un prédécesseur en utilisant les opérateurs d'accès aux fils stockés dans le paramètre `access`.
3. Modifier vos fonctions `const BinarySearchTree* bstree_successor(const BinarySearchTree* x)` et `const BinarySearchTree* bstree_predecessor(const BinarySearchTree* x)` pour qu'elles appellent la fonction `BinarySearchTree* find_next(const BinarySearchTree* x, ChildAccessors access)` avec le bon paramètre `access` pour réaliser le traitement.

Après avoir programmé et modifié ces opérateurs, vous vérifierez que votre programme s'exécute toujours correctement.

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.

Dans le cadre d'un arbre binaire de recherche, une analyse exhaustive des configurations possibles pour la suppression d'un nœud est réalisable en raison de la faible combinatoire impliquée par la structure de données. Une fois que le nœud à supprimer a été trouvé à partir de sa clé, 3 configurations sont possibles :

1. **Suppression d'une feuille** : Il suffit de l'enlever de l'arbre puisqu'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 nœud le plus à gauche du sous-arbre droit - ci-dessous, le nœud de valeur 9) ou son prédécesseur (le nœud le plus à droite du sous-arbre gauche - ci-dessous, le nœud de valeur 6). Ce remplacement conserve la propriété des arbres binaires de recherche, sauf sur le nœud N. Mais comme il va être supprimé dans l'étape suivante, la propriété des arbres binaires de recherche sera vérifiée sur l'arbre résultant.

Il est important de constater que si le nœud N a pris la place dans l'arbre 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.

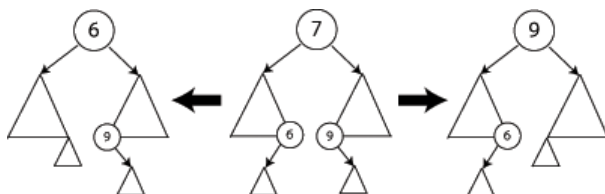


FIGURE 3 – Suppression d'un nœud dans un arbre binaire de recherche

1. Programmer la fonction `void bstree_swap_nodes(ptrBinarySearchTree* t, ptrBinarySearchTree from, ptrBinarySearchTree to);` qui échange, dans l'arbre `t` les nœuds `from` et `to`.
2. Programmer la fonction `void bstree_remove_node(ptrBinarySearchTree* t, ptrBinarySearchTree current)` qui supprime, dans l'arbre `t` le nœud `current` en suivant l'algorithme décrit ci-dessus.
Pour cette fonction si le nœud à supprimer possède deux fils, il sera échangé dans l'arbre avec son successeur avant d'être supprimé.
3. 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.
4. 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, OperateFuncion 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
        ... its successor is: 3
        ... its predecessor is: 1
    Searching for value 5 in the tree : true
        ... its successor is: 6
        ... its predecessor is: 4
    Searching for value 8 in the tree : false
```

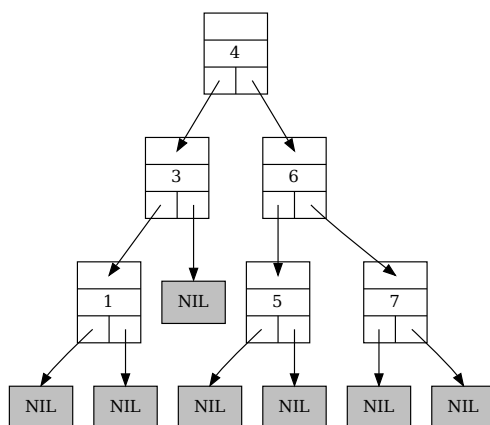
```

    Searching for value 1 in the tree : true
        ... its successor is: 2
        ... its predecessor does not exist
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 4 from the tree :      1 3 6 7
    Removing the value 3 from the tree :      1 6 7
Done.

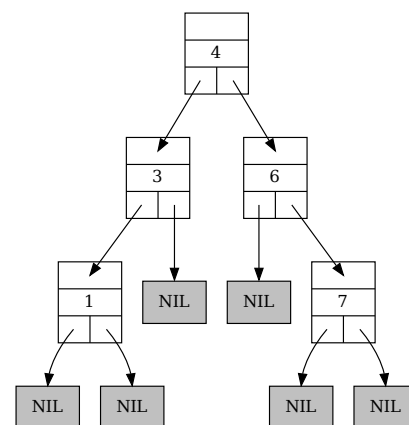
```

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`.

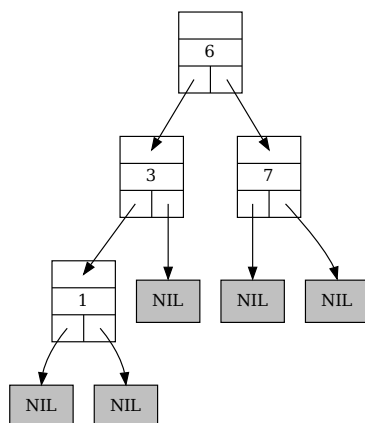
Les fichiers PDF générés montrent l'évolution de l'arbre lors des suppressions, dans l'ordre, des valeurs 2, 5, 4 et 3 à partir de l'arbre de la figure 1.



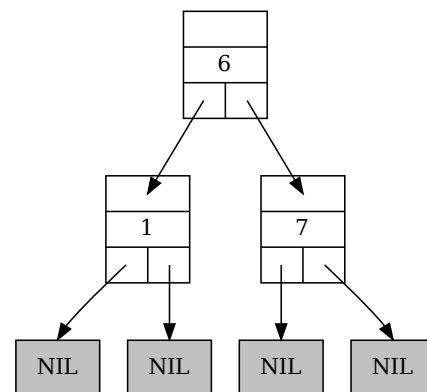
(a) Removed key : 2



(b) Removed key : 5



(c) Removed key : 4



(d) Removed key : 3

FIGURE 4 – Évolution de l'arbre de la figure 1 lors de suppressions successives.

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 (la séquence de clés suivant la clé du nœud supprimé dans la sorte de votre programme), 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 `OperateFuncutor` 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_key(bstree_iterator_value(i)));
bstree_iterator_delete(i);
```

Afin d'implanter de patron de conception pour les arbres binaires 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 précédents 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.
Searching into the tree.
    Searching for value 2 in the tree : true
        ... its successor is: 3
        ... its predecessor is: 1
    Searching for value 5 in the tree : true
        ... its successor is: 6
        ... its predecessor is: 4
    Searching for value 8 in the tree : false
    Searching for value 1 in the tree : true
        ... its successor is: 2
        ... its predecessor does not exist
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 4 from the tree :      1 3 6 7
    Removing the value 3 from the tree :      1 6 7
Done.
Iterate descending onto the tree.
    values : 7 6 1
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 justifierez le choix de visiteur que vous avez réalisé. Vous veillerez à ce qu'à 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 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.
Deleting the tree.
Done.
```