Richou D. De...

Zeste de savoir

Les arbres binaires de recherche

12 août 2019

Table des matières

Lorsque l'on parle d'arbres, bien des gens penseront de prime abord à la biologie, la botanique et donc aux arbres tels que vous pouvez sûrement les voir en jetant un coup d'œil par votre fenêtre. Cependant, les arbres se retrouvent dans un ensemble de domaines tous plus variés les uns que les autres, et c'est sur l'algorithmique que nous allons nous pencher ici. Les arbres ont effectivement inspiré les algorithmiciens de la deuxième moitié du vingtième siècle, les amenant à créer une structure de données révolutionnaire : **les arbres binaires de recherche**.

Qu'est-ce que c'est, à quoi ça sert, comment ça marche, comment les implémenter, comment les améliorer ; ce sont autant de questions qui seront abordées dans ce cours, qui va, je l'espère de tout cœur, vous brancher sans vous faire prendre racine !

0.1. Un mot sur les prérequis

Dans la mesure où ce cours traite d'algorithmique, **une connaissance basique dans ce domaine** sera nécessaire pour ne pas se perdre dans les explications. **Une bonne connaissance du C** sera un avantage à qui veut suivre les explications sur l'implémentation proposée, mais n'est pas requise à proprement parler. De plus, dans la mesure où ce cours se veut une introduction, aucune connaissance sur les arbres binaires ou sur la théorie des graphes en général n'est nécessaire.
D'une manière générale, ce cours s'adresse à quiconque a déjà touché de près ou de loin au monde de la programmation et/ou de l'algorithmique.

0.2. Remarque de vocabulaire

Par la suite, j'utiliserai sans distinction les expressions « arbres », « arbres binaires » ou « BST »[1]. Tous ces termes font bien entendu référence aux « arbres binaires de recherche », ne vous inquiétez pas.

1. Définitions

1.1. Structure de données

Comme mentionné dans l'introduction, les BST font partie de la grande famille des *structures de données*. Mais qu'est-ce donc que ces bêtes là ? Pour répondre à cette question, Wikipédia nous fournit un bon point de départ :

> Une structure de données est une structure logique destinée à contenir des données, afin de leur donner une organisation permettant de simplifier leur traitement.
>
> *Wikipédia*

Il existe de très nombreuses structures de données différentes : enregistrements, listes, matrices, piles, files, tables ou encore... arbres. Toutes ont leur propre logique, leur propre manière de fonctionner et leurs propres avantages et inconvénients. Je vous laisse vous documenter sur ces nombreux sujets si le cœur vous en dit.

1.2. Les arbres binaires de recherche

Nous allons tâcher maintenant de définir précisément ces fameux arbres binaires. Pour se donner une idée, voici un exemple :

FIGURE 1. – Un bonzaï binaire de recherche

1.2.1. Arbre

On retrouve dans la vie de tous les jours de nombreuses arborescences : les dossiers d'un ordinateur, l'organisation d'une entreprise, *et cætera*. De manière un peu plus formelle, un arbre est un ensemble d'éléments appelés **nœuds**, parmi lesquels on trouve une unique **racine** : c'est le nombre 10 dans l'exemple. Chaque nœud est relié à ses descendants grâce à des arêtes. Les

1. *Binary Search Tree*, pour les amateurs de la langue de Shakespeare.

nœuds sans aucun descendants sont appelés **feuilles** ; dans l'exemple, il y a trois feuilles : 2, 13 et 17. Pour poursuivre avec le vocabulaire lié aux arbres, on appelle **profondeur** d'un nœud « l'étage » du nœud en question[2]. Par exemple, le nœud contenant 5 a une profondeur égale à deux. Enfin, la **hauteur** désigne la profondeur la plus grande atteinte dans l'arbre. Ici, la hauteur de l'arbre est quatre : il s'agit de la profondeur des nœuds 13 et 17.

1.2.2. Binaire

La particularité d'un arbre *binaire* est que chaque nœud a au maximum deux fils : un à gauche et un à droite. Cette propriété a de nombreuses conséquences :

— On peut déceler dans les arbres binaires une structure récursive ; en effet, un arbre binaire est essentiellement composé d'un nœud, d'un sous-arbre gauche et d'un sous-arbre droit, chaque sous-arbre étant à son tour un arbre composé d'un nœud et de deux sous-arbres et ainsi de suite, jusqu'aux feuilles dont les sous-arbres sont vides.
— Chaque nœud ne possédant *au maximum* que deux descendants, un arbre de hauteur h peut contenir au maximum $2^h - 1$ nœuds. Inversement, un arbre contenant n nœuds a une hauteur minimale de $\lceil \log_2(n) \rceil$.

1.2.3. De recherche

Les éléments stockés dans un arbre binaire de recherche doivent posséder une relation d'ordre totale , c'est-à-dire qu'il doit exister une manière de dire si un élément est plus grand ou plus petit qu'un autre. Par exemple, on peut trier des nombres par valeur ou des personnes par ordre alphabétique de leur nom de famille.

Le premier élément inséré dans l'arbre devient la racine. Ensuite, il suffit de mettre à gauche les éléments plus petits et à droite les éléments plus grands. C'est cette particularité qui rend les BST intéressants : la plupart des opérations réalisées sur les arbres binaires de recherche ont une complexité temporelle logarithmique ($O(\log n)$) dans le cas moyen. Cela est dû au fait que, grâce à la relation d'ordre liant les valeurs, on peut naviguer dans l'arbre avec une logique rappelant une recherche dichotomique , ce qui est **plus performant** que les listes, par exemple, que l'on doit parcourir élément par élément[3].

1.3. Exemple

Afin de tirer les choses au clair, voici un exemple d'utilisation des arbres binaires de recherche, afin que vous puissiez les voir en action. Un vétérinaire voudrait stocker les fiches médicales de ses patients, et, plutôt que d'utiliser un tableau ou une liste, on se propose d'utiliser un arbre binaire. La fiche contiendra différentes informations sur l'animal ; on utilisera son nom comme **clé** (c'est-à-dire comme critère pour la relation d'ordre), que l'on triera selon l'ordre alphabétique croissant.

Le vétérinaire reçoit sa première patiente, qui répond au nom de **Gaufrette**. Comme sa fiche sera le premier nœud de notre arbre, elle en devient automatiquement la racine.

2. Attention, on considère que la racine a une profondeur égale à 1.
3. Ce genre d'algorithmes amènent à une complexité linéaire ($O(n)$).

FIGURE 1. – Un peu seule, cette racine...

Après Gaufrette, vient le tour de **Charlie**. Une fois sa fiche remplie, on va l'ajouter à l'arbre : comme *Charlie* est avant *Gaufrette* dans l'ordre alphabétique, on place la fiche de Charlie à gauche de la racine.

FIGURE 1. – Charlie est avant Gaufrette

Le prochain patient est **Médor**. Vous l'aurez compris, *Médor* vient après *Gaufrette*, on placera donc sa fiche à droite de la racine.

FIGURE 1. – Médor est après Gaufrette

Allez, c'est à vous de jouer à présent ! Les trois prochains patients s'appellent **Flipper**, **Bubulle** et **Augustin**, et ils sont soignés par le vétérinaire dans cet ordre. Prenez quelques instants pour deviner à quoi ressemble l'arbre après ces nouvelles consultations. La solution se trouve juste après.

Contenu masqué n°1

La structure d'un arbre est dépendante de l'ordre dans lequel on insère les éléments ! Si on avait ajouté *Augustin* avant *Bubulle*, l'arbre aurait été comme ceci :

FIGURE 1. – Ils se ressemblent, mais ne sont pas identiques !

Nous reviendrons bien plus tard sur les implications de cette propriété, mais sachez que c'est très loin d'être anodin.

2. Opération et implémentation

Le but ici est de détailler l'implémentation des opérations sur les arbres binaires. Les algorithmes seront détaillés en pseudo-code par souci de généralisme, et une implémentation en C sera proposée afin d'approcher les algorithmes d'un point de vue un peu plus technique. Bien sûr, **rien ne vous empêche de suivre le cours si vous ne connaissez pas le C**, voire même de publier votre propre bibliothèque dans le langage de votre choix ! Comme toujours sur Zeste de Savoir, toute contribution est la bienvenue !

Une implémentation complète, fonctionnelle et commentée en C est disponible ici .

2.1. Définition de la structure

Ne mettons pas la charrue avant les bœufs : avant de penser aux fonctions destinées à être utilisées sur des BST, il faut peut-être commencer par définir les BST eux-mêmes, non ?

2.1.1. Pseudo-code

Du point de vue algorithmique, un BST n'a rien de très compliqué. Un nœud contient une donnée quelconque, un nœud fils à gauche, et un nœud fils à droite. On retrouve la structure récursive évoquée précédemment.

```
Structure bst_node :
    data est de type quelconque
    left est de type bst_node
    right est de type bst_node
```

On peut remarquer que les champs `left` et `right` peuvent être vides : dans le cas d'une feuille, par exemple, le nœud contient une donnée, mais n'a ni fils gauche, ni fils droit.

Enfin, pour définir l'arbre, on n'a besoin de connaître une seule chose : **sa racine**. En effet, une fois que la racine est connue, on peut accéder à ses successeurs, puis aux successeurs de ses successeurs, et ainsi de suite...

2.1.2. C

En C, en revanche, c'est un peu plus compliqué. Voici la structure qui sera utilisée pour les implémentations à suivre :

Contenu masqué n°2

Pour que la bibliothèque soit générique (c'est-à-dire qu'elle fonctionne quelque soit le type des données stockées dans l'arbre), le C nous propose le type `void*`. Cependant, on a besoin de connaître la taille d'une donnée (appelée `data_size` dans la structure C), ainsi que la fonction à utiliser pour réaliser la relation d'ordre, désignée par `compare` dans la structure.

La notation suivante

```c
typedef int (*orderFun)(void *, void *);
```

permet de déclarer **un type** `orderFun` désignant une fonction renvoyant un entier et prenant comme arguments deux `void*`. Lorsque l'arbre devra réaliser une comparaison, il appellera la fonction `compare` et déclarera que le premier argument est supérieur au second si le résultat est positif (ou nul), ou que le second argument est supérieur au premier si le résultat est négatif.
On retrouve ce fonctionnement dans les fonctions standard du C, comme `strcmp` .

Enfin, on retrouve la fonction `free` de signature `void free(void *)`. Cette fonction sera utilisée lors de la désallocation d'un nœud (lors d'une suppression ou de la destruction de l'arbre, par exemple) de sorte à éviter toute fuite de mémoire , dans le cas où les nœuds de l'arbre contiendrait des données allouées dynamiquement.

Comme vous le voyez, tout cela se révèle être beaucoup plus complexe que la structure algorithmique de l'arbre. Néanmoins, on peut écrire une fonction afin de rendre sûr et transparent la création d'un arbre, regardez :

Contenu masqué n°3

La fonction `assert` [4] nous permet d'interrompre le programme avec un message d'erreur si la condition donnée est fausse. En effet, laisser un programme s'exécuter alors que le paramètre `data_size` ou la relation d'ordre de l'arbre sont incohérents est une mauvaise idée.
On peut également utiliser la fonction `perror` pour afficher un message d'erreur si l'allocation dynamique `malloc` a rencontré un problème et quitter le programme avec un code d'erreur en utilisant `exit(EXIT_FAILURE)`[5].

2.2. Ajout d'un nœud

2.3. Pseudo-code

Comme nous l'avons vu précédemment on peut ajouter un nouveau nœud à l'arbre très simplement à l'aide d'une fonction récursive :

```
1  Procédure bst_add (node de type bst_node, element de type
       quelconque) :
2      Si node est vide
3          node.data := element
4      Sinon
5          Si node.data > element
6              bst_add(node.left, element)
7          Sinon
8              bst_add(node.right, element)
9          FinSi
10     FinSi
```

Comme le nœud `node.left` (respectivement `node.right`) de la racine peut être considéré comme la racine du sous-arbre gauche (droit), on peut utiliser la fonction elle-même pour réaliser l'insertion dans le sous-arbre, et répéter l'opération jusqu'à atteindre un sous-arbre vide, où l'on vient insérer la nouvelle donnée.

2.3.1. C

Ici, nous allons utiliser exactement la même méthode. Néanmoins, les structures pseudo-code et C étant différentes, les fonctions le seront un peu aussi. Regardons pour commencer la signature de la fonction `bst_add` en C :

4. Qui est en fait une macro, mais chut !
5. `EXIT_FAILURE` est une constante de préprocesseur définie dans l'en-tête `stdlib.h` et a une valeur non nulle, sachant que le standard C veut qu'un programme se terminant correctement renvoie 0, tout autre valeur correspondant à une erreur.

```c
/**
 * \brief         Adds a new node to a tree
 *
 * Calling this function dynamically allocates a new node and
    inserts it into
 * the binary search tree.
 *
 * \param         tree         The tree to insert the new node into.
    Cannot be @c NULL
 * \param         element         The new node's data. Cannot be @c
    NULL
 */
void bst_add(bst* tree, void* element);
```

A priori, rien de très différent par rapport au pseudo-code précédent. Et pourtant, le premier argument est un arbre, et non pas un nœud, ce qui signifie que l'on ne va pas pouvoir utiliser la récursivité directement dans la fonction `bst_add`. Embêtant, non ? Une technique valable dans ce genre de situation est de créer le nœud contenant la nouvelle donnée dans `bst_add` puis d'ajouter le nouveau nœud dans l'arbre grâce à une fonction récursive avec une signature comme suit :

```c
static void bst_add_rec(orderFun compare, bst_node* current,
    bst_node* new);
```

Les fonctions statiques
Le mot-clé `static` présent devant la signature de la fonction signifie que seules les fonctions de la même *unité de compilation*[6] peuvent y faire appel. En clair, cela signifie que l'utilisateur de notre bibliothèque n'aura pas le droit d'utiliser directement `bst_add_rec`, ce qui est une bonne chose.

Jetons un œil aux arguments de la fonctions `bst_add_rec` :

- `compare` : la fonction `bst_add_rec` va avoir besoin de réaliser des comparaisons pour savoir de quel côté d'un nœud se diriger ; on va donc lui envoyer la relation d'ordre de l'arbre.
- `current` : ce paramètre désigne la racine de l'arbre (ou du sous-arbre) dans lequel on veut insérer le nouveau nœud.
- `new` : ici, ce sera le nœud à insérer, qui aura été préalablement créé dans `bst_create_node`, une simple fonction auxiliaire.

Si tout est clair pour tout le monde, voici l'implémentation des fonctions `bst_add` et `bst_create_node` pour commencer :

6. Une unité de compilation correspond en règle générale à un fichier source après passage du préprocesseur.

Contenu masqué n°4

Dans `bst_create_node`, on alloue de l'espace pour le nœud ainsi que pour la donnée à stocker, on copie la donnée dans l'espace que l'on vient de créer, et on déclare que le nouveau nœud n'a pas de descendants. Vient ensuite un test permettant de vérifier que la racine de `tree` existe. En effet, s'il s'agit du premier nœud de l'arbre, `tree->root` vaut `NULL`, et l'ajout se résume à une simple affectation. Dans le cas contraire, on déclenche la récursivité en insérant le nouveau nœud à partir de la racine déjà existante.

Jetons maintenant un œil à `bst_add_rec` :

Contenu masqué n°5

La lecture de ce code est plutôt directe : on cherche d'abord à savoir si on se dirige sur la gauche ou la droite du nœud `current`, puis si le sous-arbre correspondant existe ou non. Si oui, on insère le nouveau nœud dans le sous-arbre par récursivité ; sinon, on ajoute le nœud par affectation.

Fonction de destruction
On implémente les nœuds et les données de manière dynamique ici. Il est donc très important de prévoir leur désallocation. On peut utiliser une fonction récursive avec la même ruse vue précédemment. On peut écrire une petite fonction `bst_destroy_node` pour s'aider :

Contenu masqué n°6

On utilise `bst_destroy` pour lancer la récursivité, puis, dans `bst_destroy_rec`, on désalloue le sous-arbre gauche s'il existe, le sous-arbre droit s'il existe, puis la racine grâce à `bst_destroy_node`.

2.4. Parcours

Jusqu'à présent, nous sommes capables de créer des arbres binaires, et d'y ajouter des données. C'est déjà pas mal, mais il serait autrement plus intéressant de pouvoir exploiter les données stockées dans l'arbre. Nous allons donc écrire une fonction nous permettant de parcourir tous les nœuds de l'arbre et de leur appliquer une procédure[7] donnée.

7. Une *procédure* est en fait une fonction qui ne renvoie pas de résultat. En C, une procédure est donc une fonction ayant `void` comme type de retour.

2.4.1. Pseudo-code

Ici, rien de plus simple. On va simplement appliquer `procedure` à tous les nœuds de l'arbre par récursivité :

```
Procédure bst_iter (node de type bst_node, procedure de type
    procédure) :
    Si node est vide
        Quitter la procédure
    Sinon
        bst_iter(node.left, function)
        procedure(node.data)
        bst_iter(node.right, function)
    FinSi
```

Traversée *pre-order*, *in-order*, *post-order*
Dans le pseudo-code ci-dessus, j'ai arbitrairement choisi de mettre la ligne `procedure(node.data)` *entre* l'appel récursif sur le sous-arbre gauche et celui sur le droit. On parle de traversée *in-order* ; en effet, dans le cas des BST, cette traversée a la particularité de visiter tous les nœuds *dans l'ordre*, puisqu'on visite d'abord le sous-arbre gauche dont les valeurs sont inférieures, puis le nœud lui-même, et enfin le sous-arbre droit qui contient les nœuds supérieurs.

FIGURE 2. – Traversée in-order : 2, 5, 10, 12, 13, 15, 17

Il existe aussi d'autres types de parcours :

— ***pre-order*** : on parcourt d'abord le nœud, puis le sous-arbre gauche, puis le droit.

FIGURE 2. – Traversée pre-order : 10, 5, 2, 12, 15, 13, 17

— ***post-order*** : on parcourt le sous-arbre gauche, puis le droit, et enfin le nœud.

FIGURE 2. – Traversée post-order : 2, 5, 13, 17, 15, 12, 10

2.4.2. C

Comme toujours, nous allons devoir utiliser deux fonctions : une fonction pour déclencher la récursivité, et une fonction statique récursive. On définit tout de même le type `iterFun` par souci de lisibilité : la fonction `function` doit prendre comme argument un `void*` (un pointeur sur la donnée) et ne rien retourner.

```c
typedef void (*iterFun)(void *);
```

Contenu masqué n°7

2.4.3. Exemple

Entre la création, la destruction, l'ajout d'un nœud et la traversée, notre petite bibliothèque commence à se remplir. Elle l'est même suffisamment pour écrire un premier programme de test ! Si vous avez bien tout suivi depuis le début, tout devrait vous paraître logique :

Contenu masqué n°8

On commence par définir la fonction `comp` dont l'arbre va avoir besoin pour fonctionner. La syntaxe `*((int*) a) - *((int*) b)` est un peu surprenante de prime abord, puisqu'on a besoin de déréférencer les arguments préalablement castés en `int*`, mais en gros, cette fonction ne fait que renvoyer $a - b$, qui sera positif si $a > b$ et négatif si $b > a$. Exactement ce dont notre arbre a besoin. La fonction `print` quant à elle ne fait qu'afficher le nombre qui lui est envoyé. On utilise cette fonction avec `bst_iter` pour afficher toutes les valeurs contenues dans l'arbre.

Si tout se passe bien, voici l'affichage que tout un chacun doit obtenir :

```
Creating tree...
Done.
Adding 10...
Adding 5...
Adding 2...
Adding 12...
Adding 15...
Adding 17...
Adding 13...
```

```
Printing data...
2        5        10        12        13        15        17        Done.
Destroying tree...
Done.
```

2.5. Recherche

Une autre fonction couramment implémentée dans les structures de données permet de rechercher un élément. Dans notre cas, la fonction peut renvoyer le nœud dans lequel l'élément recherché a été trouvé, et une valeur par défaut (notée `(nil)` dans le pseudo-code) si la valeur n'a pas été trouvée.

2.5.1. Pseudo-code

Comme tout à l'heure avec `bst_add`, on va utiliser des comparaisons pour savoir si on continue la recherche à gauche ou à droite d'un nœud. Et si d'aventure, un élément n'est ni supérieur ni inférieur, c'est qu'il correspond à l'élément que l'on cherche.

```
Fonction bst_search(node de type bst_node, element de type
    quelconque) :
    Si node est vide
        Renvoyer (nil)
    FinSi
    Si node.data > element
        Renvoyer bst_search(node.left, element)
    Sinon Si node.data < element
        Renvoyer bst_search(node.right, element)
    Sinon
        Renvoyer node
    FinSi
```

2.5.2. C

A ce stade, vous devriez avoir compris le principe. Nous allons écrire une fonction `bst_search` qui va simplement s'assurer que l'arbre existe et qui va ensuite déclencher la récursivité à partir de la racine en appelant une fonction récursive `bst_search_rec`, qui elle va beaucoup ressembler au pseudo-code. Voici ce à quoi cela pourrait ressembler :

Contenu masqué n°9

2.6. Suppression

La suppression d'un nœud de l'arbre est sans aucun doute l'opération la plus compliquée que nous verrons ici. En effet, la fonction de suppression doit, en plus de supprimer le nœud, faire le nécessaire pour conserver la structure de l'arbre, et c'est loin d'être simple.

2.6.1. Pseudo-code

Lors de la suppression d'un nœud, il y a trois cas différents à considérer :

— **Le nœud n'a aucun descendant** : pas de problème ici, on peut supprimer le nœud directement.

FIGURE 2. – Suppression d'un nœud sans descendants

— **Le nœud a un descendant (gauche ou droit)** : avant de supprimer le nœud, il faut rattacher son père à son descendant. A part ça, tout va bien.

FIGURE 2. – Suppression d'un nœud avec un descendant (ici à gauche)

— **Le nœud a deux descendants (gauche et droit)** : là, c'est la galère. On ne peut pas remplacer le nœud par son fils gauche, car il faudrait replacer tous les nœuds à droite de ce fils de l'autre côté (cf. diagramme ci-dessous), ce qui peut se révéler atrocement long si l'arbre contient ne serait-ce que quelques milliers de nœuds. Dans l'exemple ci-dessus, on pourrait parfaitement essayer de supprimer 10. Cependant, la méthode à employer pour conserver la structure de l'arbre est loin d'être évidente.

FIGURE 2. – Si on veut supprimer le 10 de l'arbre, comment faire ?

La solution consiste à trouver le **successeur** du nœud que l'on veut supprimer. Le successeur d'un nœud est en fait le nœud le plus grand (c'est-à-dire le plus à droite) du sous-arbre gauche[8]. En effet, le successeur est par définition le plus grand nœud inférieur à celui qui doit être supprimé, donc on peut remplacer le nœud à supprimer par son successeur, et le tour est joué ! Pour clarifier tout ça, rien ne vaut un bon diagramme :

FIGURE 2. – Compliqué, mais on y arrive !

Maintenant, se pose la question de l'algorithme. On décompose la suppression d'un nœud en deux étapes ; premièrement, on recherche le nœud à supprimer, puis on réalise la suppression en elle-même, en fonction du cas dans lequel on se trouve.

```
1  Fonction bst_remove(node de type bst_node, element de type
       quelconque) :
2      Si node est vide
3          Renvoyer (nil) // On a pas trouvé d'élément à supprimer
4      FinSi
5      Si node.data > element
6          node.left = bst_remove(node.left, element) // L'élément à
               supprimer se trouve dans le sous-arbre gauche
7      Sinon Si node.data < element
8              node.right = bst_search(node.right, element) //
                   L'élément à supprimer se trouve dans le sous-arbre
                   droit
9      Sinon // On a trouvé l'élément à supprimer
10         Si node.left est vide ET node.right est vide // Aucun fils
               : Cas n°1
11             Renvoyer (nil)
12         Sinon Si node.right est vide // Un seul fils à gauche : Cas
               n°2
13             Renvoyer node.left
14         Sinon Si node.left est vide // Un seul fils à droite : Cas
               n°2
15             Renvoyer node.right
16         Sinon // Un fils à gauche et à droite : Cas n°3
17             successor = Max(node.left) // Ou Min(node.right)
18             node.data = successor.data
19             bst_remove(node.left, successor.data) // On supprime le
                   successeur par récursivité
20         FinSi
21     FinSi
```

8. Le nœud le plus petit (le plus à gauche) du sous-arbre droit est aussi un successeur.

```
22      Renvoyer node
```

La fonction `Max`

Rechercher le maximum d'un arbre est très facile : il s'agit du premier nœud qui ne possède pas de fils droit.

```
1 Fonction Max(node de type bst_node) :
2     Si node.right est vide
3         Renvoyer node
4     Sinon
5         Renvoyer Max(node.right)
6     FinSi
```

On peut de la même manière écrire une fonction `Min` qui recherche le nœud le plus à gauche de l'arbre.

Vous l'aurez bien compris, cette fonction est de très loin la plus compliquée qu'il vous sera donné d'étudier dans ce cours. En plus de ça, il y a encore un détail qui mérite d'être éclairci. Lorsque l'on supprime un nœud, il faut mettre à jour son père, soit pour le lier au fils du nœud supprimé (cas 2) soit pour lui signaler qu'il n'a plus de fils (cas 1).

Seulement, vous aurez sans doute remarqué que les arêtes sur tous les diagrammes de ce cours sont des *flèches* : un nœud `node` connaît ses descendants, `node.left` et `node.right`, **mais il ne connaît pas son père**. Pour palier à cette difficulté, on fait en sorte que la fonction `bst_remove` renvoie une valeur : **la nouvelle racine de l'arbre**. Comme ça, la ligne suivante

```
1 node.left = bst_remove(node.left, element)
```

permet de mettre à jour `node.left` en fonction de se qui se passera dans `bst_remove(node.left, element)`.

2.6.2. C

Le pseudo-code de la suppression est assez compliqué, pas vrai ? Ça tombe bien, parce que le C n'apportera aucune difficulté supplémentaire, pour une fois. On doit simplement penser à désallouer l'espace mémoire du nœud que l'on supprime. On peut utiliser la fonction `bst_destroy_node` évoquée plus tôt.

Contenu masqué n°10

Et voilà le travail ! On dispose ici d'une interface classique pour gérer des BST depuis n'importe quel type de programme ! Encore une fois, n'hésitez pas à proposer vos travaux, qu'il s'agisse

d'implémentations de cette bibliothèque dans le langage de votre choix, une amélioration de celle proposée, ou même un projet qui utilise cette bibliothèque, **toutes vos contributions sont les bienvenues** !

3. Inconvénients et évolutions

3.1. Déséquilibre et dégénérescence en liste

Malheureusement, les arbres binaires de recherche ne possèdent pas que des avantages[9]. Pour s'en rendre compte, il suffit d'imaginer un arbre dans lequel on insère successivement 1, 2, 3, 4 et 5.

FIGURE 3. – Plus très binaire, cet arbre binaire...

Eh oui, les arbres binaires de recherche sont sensibles à ce que l'on appelle la **dégénérescence en liste**. Dans l'exemple ci-dessus, on perd tout l'intérêt des arbres puisque l'on est revenu à une autre structure de données (une liste chaînée pour être exact) qui ne bénéficie pas des avantages des arbres.

De manière plus générale, la dégénérescence en liste d'un arbre est un cas extrême de **déséquilibre**. Un arbre est déséquilibré à partir du moment où un niveau de l'arbre (sauf le dernier) n'est pas rempli.

FIGURE 3. – Tous les niveaux sont pleins (sauf le dernier) : l'arbre est équilibré.

FIGURE 3. – L'avant-dernier niveau n'est pas plein : l'arbre est déséquilibré.

9. Ça serait trop beau.

Un déséquilibre provoque un rallongement du temps moyen des opérations, donc on cherche évidemment à éviter cette situation. Malheureusement, avec l'implémentation que nous avons vu durant ce cours, il est difficilement possible de remédier à un tel problème.

3.2. Les arbres AVL

En 1962, les algorithmiciens Adelson-Velsky et Landis publient leurs recherches sur une sorte « d'arbre binaire de recherche évolué », que l'on appelle aujourd'hui **arbres AVL** en référence aux initiales de leurs inventeurs. La différence entre les BST classiques et les arbres AVL est simple : les AVL sont capables de s'**auto-équilibrer**. Ils préservent donc les qualités des arbres binaires en termes de performances, et les exploitent au maximum puisqu'un AVL n'est jamais déséquilibré.

Cette évolution s'appuie sur une nouvelle opération : **la rotation**.

> *i* Le but de ce qui n'est va suivre n'est pas de réaliser l'implémentation des AVL, donc aucun pseudo-code ou C ne sera proposé ici.

À chaque fois que l'on modifie la structure de l'arbre (c'est-à-dire après l'ajout ou la suppression d'un nœud), on doit vérifier si l'opération a déséquilibré l'arbre ou pas. Pour ce faire, on calcule la **balance** de tous les ancêtres du nœud qui a été modifié, avec la balance définie comme suit :

$$\text{balance} = \text{hauteur du sous-arbre gauche} - \text{hauteur du sous-arbre droit}$$

D'après cette définition, dans un arbre équilibré, toutes les balances sont comprises entre -1 et 1 (inclus). Si la balance d'un nœud sort de cet intervalle, cela signifie que l'on a détecté un déséquilibre. Pour le résoudre, on effectue la rotation comme suit :

FIGURE 3. – Illustration d'une rotation, après l'ajout de 17, par exemple

On parvient ainsi à garder un arbre équilibré à tout moment.

3.3. Autres évolutions

Les arbres AVL ne sont pas les seules variantes des arbres binaires de recherche. Parmi les autres évolutions des BST, on retrouve notamment les arbres bicolores , les arbres-tas , ou encore les arbres splay . Je vous invite de tout cœur à vous documenter sur ces structures, qui constitueront un excellent complément au cours de vous venez de suivre.

Voilà, ce cours touche à sa fin ; j'espère qu'il vous aura été aussi instructif que possible. Je vous encourage une nouvelle fois à réaliser vous-même l'implémentation des BST dans le langage de votre choix. En plus de mettre en pratique tout ce que vous aurez appris ici, vous participerez à l'établissement d'une petite banque de code permettant à tout un chacun d'aborder ce cours avec son langage privilégié ou avec un langage qu'il souhaite découvrir à travers un exemple concret.

Merci à tous pour votre lecture, et à très bientôt !

Contenu masqué

Contenu masqué n°1

FIGURE 3. – Et ainsi de suite...

Retourner au texte.

Contenu masqué n°2

```
1  typedef int (*orderFun)(void *, void *);
2  typedef void (*freeFun)(void *);
3
4  typedef struct _bst_node {
5          void* data;          /**< The node's data */
6          struct _bst_node* left;          /**< The node's left
               sub-tree */
7          struct _bst_node* right;          /**< The node's right
               sub-tree */
8  } bst_node;
9
10 typedef struct {
```

```
11          size_t data_size; /**< Size of an element */
12          bst_node* root; /**< Root of the bst */
13          orderFun compare; /**< Binary relation of the tree dataset
                */
14          freeFun free; /**< Dynamic deallocation of tree data */
15  } bst;
```

Retourner au texte.

Contenu masqué n°3

```
 1  /**
 2   * \brief         Creates a tree
 3   *
 4   * Use this function to create a new tree. Make sure you respect
 5   * the following conditions since some @c asserts are used to check
        the
 6   * parameters relevance.
 7   *
 8   * To deallocate the tree, use @ref bst_destroy
 9   *
10   * \param         data_size         The size of an element; must be
        > 0.
11   * \param         orderFun          The tree's binary relation;
        cannot be @c NULL.
12   * \param         freeFun          The tree's data freeing function;
        can be NULL.
13   *
14   * \return         A pointer to the allocated tree
15   *
16   * \warning         Breaking any of the above conditions will cause
        the program to
17   * abort!
18   * \warning         Not deallocating the tree with @ref bst_destroy
        will result in
19   * memory leaks!
20   *
21   * \see         bst_destroy
22   */
23  bst* bst_new(size_t data_size, orderFun compare, freeFun free) {
24          assert(data_size > 0);
25          assert(compare);
26
27          bst* tree = malloc(sizeof(bst));
28          if(!tree) {// If allocation failed
29                  perror("malloc");
```

```
30                  exit(EXIT_FAILURE);
31          }
32          tree->data_size = data_size;
33          tree->compare = compare;
34          tree->free = free;
35          tree->root = NULL;
36
37          return tree;
38  }
```

Retourner au texte.

Contenu masqué n°4

```
1   void bst_add(bst* tree, void* element) {
2           assert(tree);
3           assert(element);
4
5           // Creating the new node
6           bst_node* node = bst_create_node(tree->data_size, element);
7
8           if(tree->root) // If the tree already has a root
9                   bst_add_rec(tree->compare, tree->root, node); //
                        Start recursion
10          else
11                  tree->root = node; // Setting the new node as the
                        tree's root
12  }
13
14  static bst_node* bst_create_node(int data_size, void* element) {
15          bst_node* node = malloc(sizeof(bst_node)); // Allocating a
                node
16          if(!node) {
17                  perror("malloc");
18                  exit(EXIT_FAILURE);
19          }
20          node->data = malloc(data_size); // Allocating data
21          if(!(node->data)) {
22                  perror("malloc");
23                  exit(EXIT_FAILURE);
24          }
25          memcpy(node->data, element, data_size); // Setting data
26          node->left = NULL;
27          node->right = NULL;
28
29          return node;
```

```
30 }
```

Retourner au texte.

Contenu masqué n°5

```
1  static void bst_add_rec(orderFun compare, bst_node* current,
       bst_node* new) {
2          if(compare(current->data, new->data) >= 0) { // current >=
               new
3                  if(!current->left) // current->left == NULL
4                          current->left = new;
5                  else
6                          bst_add_rec(compare, current->left, new);
7          } else { // current < new
8                  if(!current->right)
9                          current->right = new;
10                 else
11                         bst_add_rec(compare, current->right, new);
12         }
13 }
```

Retourner au texte.

Contenu masqué n°6

```
1  /**
2   * \brief         Destroys a tree
3   *
4   * A call to bst_destroy deallocates a tree and all its nodes by
        calling
5   * bst#free. Make sure the program calls it when the tree is not
        needed anymore.
6   *
7   * \param         tree         The tree to deallocate
8   *
9   * \warning         Not using this function to deallocate a bst
        will cause memory
10  * leaks!
11  * */
12 void bst_destroy(bst* tree) {
13         assert(tree);
14         if(tree->root)
```

```
15                  bst_destroy_rec(tree, tree->root);
16          free(tree);
17  }
18
19  static void bst_destroy_rec(bst* tree, bst_node* current) {
20          if(current->left)
21                  bst_destroy_rec(tree, current->left);
22          if(current->right)
23                  bst_destroy_rec(tree, current->right);
24          bst_destroy_node(tree, current);
25  }
26  static void bst_destroy_node(bst* tree, bst_node* node) {
27          if(tree->free) // If the tree has a deallocation
                function...
28                  tree->free(node->data); // ... free the node's data
                        dynamic parts
29          free(node->data); // Free the node's data slot itself
30          free(node); // Free the node
31  }
```

Retourner au texte.

Contenu masqué n°7

```
1   /**
2    * \brief         Iterates over a tree
3    *
4    * Applies a function to every element in the tree.
5    *
6    * \param         tree         The tree to iterate over
7    * \param         function         The function to apply. Cannot be
        @c NULL.
8    *
9    * \warning         Breaking any of the above conditions will cause
        the program to
10   * abort!
11   */
12  void bst_iter(bst* tree, iterFun function) {
13          assert(tree);
14          assert(function);
15          if(tree->root)
16          bst_iter_rec(tree->root, function);
17  }
18
19  static void bst_iter_rec(bst_node* current, iterFun function) {
20          if(!current)
```

```
21                  return;
22          bst_iter_rec(current->left, function);
23          function(current->data);
24          bst_iter_rec(current->right, function);
25  }
```

Retourner au texte.

Contenu masqué n°8

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include "bst.h" // Don't forget that bit
4
5   /*
6    * Comparison function
7    */
8   int comp(void* a, void* b) {
9           return *((int*) a) - *((int*) b); // (a-b)>0 if a>b, <0
                otherwise
10  }
11
12  /*
13   * Printing function
14   */
15  void print(void* a) {
16          printf("%d\t", *((int*) a));
17  }
18
19  int main(int argc, char* argv[]) {
20          bst* tree;
21          int value = 10;
22
23          printf("Creating tree...\n");
24          tree = bst_new(sizeof(int), comp, NULL);
25          printf("Done.\n");
26
27          printf("Adding %d...\n", value);
28          bst_add(tree, &value);
29
30          value = 5;
31          printf("Adding %d...\n", value);
32          bst_add(tree, &value);
33
34          value = 12;
35          printf("Adding %d...\n", value);
```

```
36          bst_add(tree, &value);
37
38          value = 15;
39          printf("Adding %d...\n", value);
40          bst_add(tree, &value);
41
42          value = 17;
43          printf("Adding %d...\n", value);
44          bst_add(tree, &value);
45
46          value = 13;
47          printf("Adding %d...\n", value);
48          bst_add(tree, &value);
49
50          printf("Printing data...\n");
51          bst_iter(tree, print);
52          printf("Done.\n");
53
54          printf("Destroying tree...\n");
55          bst_destroy(tree);
56          printf("Done.\n");
57
58          return EXIT_SUCCESS;
59  }
```

Retourner au texte.

Contenu masqué n°9

```
1   /**
2    * \brief         Searches a tree
3    *
4    * Searches tree for element. A node is considered a result
       whenever bst#compare
5    * returns 0.
6    *
7    * \param         tree         The tree to search. Cannot be @c
       NULL.
8    * \param         element         The element to search for
9    *
10   * \return         A pointer to the node element was found, @c NULL
       if element
11   * couldn't be found.
12   *
13   * \see         bst#compare
14   * \see         orderFun
```

```
 **/
bst_node* bst_search(bst* tree, void* element) {
        assert(tree);
        if(tree->root)
                return bst_search_rec(compare, tree->root,
                    element);
        return NULL;
}

static bst_node* bst_search_rec(orderFun compare, bst_node*
    current, void* element) {
        if(!current)
                return NULL;
        if(compare(current->data, element) > 0) // current >
            element
                return bst_search_rec(compare, current->left,
                    element);
        else if(compare(current->data, element) < 0)  // current <
            element
                return bst_search_rec(compare, current->right,
                    element);
        else // current == element
                return current;
}
```

Retourner au texte.

Contenu masqué n°10

```
void bst_remove(bst* tree, void* element) {
        assert(tree);
        if(tree->root)
                bst_remove_rec(tree, tree->root, element);
}

static bst_node* bst_remove_rec(bst* tree, bst_node* current, void*
    element) | {
        if(!current)
                return NULL;
        if(tree->compare(current->data, element) > 0)
                current->left = bst_remove_rec(tree, current->left,
                    element);
        else if(tree->compare(current->data, element) < 0)
                current->right = bst_remove_rec(tree,
                    current->right, element);
        else {
```

Contenu masqué

```
15                  if(!current->left && !current->right) { // No
                        children
16                          bst_destroy_node(tree, current);
17                          return NULL;
18                  }
19                  else if(!current->right) { // Only a left child
20                          bst_node* temp = current->left;
21                          bst_destroy_node(tree, current);
22                          return temp;
23                  }
24                  else if(!current->left) { // Only a right child
25                          bst_node* temp = current->right;
26                          bst_destroy_node(tree, current);
27                          return temp;
28                  }
29                  else { // Left and right children
30                          bst_node* successor =
                                find_max(current->left);
31                          memcpy(current->data, successor->data,
                                tree->data_size);
32                          current->left = bst_remove_rec(tree,
                                current->left, successor->data);
33                  }
34          }
35          return current;
36  }
37
38  static bst_node* find_max(bst_node* current) {
39          if(!current->right)
40                  return current;
41          return find_max(current->right);
42  }
```

Retourner au texte.