TP n°21 - Équilibrage des ABR

Définition 1 (Arbre binaire équilibré)

Soit $\mathcal{S} \in \mathcal{T}$ un ensemble d'arbres binaires. On dit que les arbres de S sont équilibrés s'il existe une constante c telle que, pour n'importe quel arbre $t \in \mathcal{S}$, on a :

$$h(t) \leq c \log_2(n(t)+1) - 1$$

avec notre convention sur la hauteur $h(E) = -1$.

Remarque. On peut prendre n'importe quel logarithme dans la définition ci-dessus.
c est la même constante pour TOUS les arbres de $\mathcal{S}$.
Comme on a toujours $h(t) \geq \lceil \log_2(n(t)+1) \rceil - 1$, cela revient à dire que les arbres de $\mathcal{S}$ ont une hauteur qui évolue de manière logarithmique avec leur nombre de nœuds : si j'insère des nœuds, je le ferai de manière suffisamment compacte, en évitant tout phénomène de peigne.

Soit $\mathcal{S}$ un ensemble d'arbres binaires construits selon un certain procédé. Garantir que le arbres de $\mathcal{S}$ sont équilibrés permet d'être certain que tous les arbres générés selon ce procédé auront des hauteurs contrôlées, évoluant de manière logarithmique avec le nombre de nœuds.

Attention. **Pour les ABR, tous les accesseurs et transformateurs ont une complexité temporelle directement en lien avec la hauteur de l'arbre. Si les ABR générés sont équilibrés, on sera alors assurés que toutes ces opérations s'effectueront avec une complexité au pire logarithmique par rapport au nombre de nœuds de l'arbre binaire.**

L'analyse de l'équilibre des ABR est donc essentielle pour la maîtrise de la complexité temporelle des algorithmes utilisant cette structure de données.

Commençons par étudier l'équilibre des ABR générés avec notre méthode de construction actuelle.

Exercice 1 (Analyse de l'équilibre d'ABR générés aléatoirement sans rééquilibrage).

1. Récupérer le code `bst-balance-analysis-init.c` sur Moodle, ranger le correctement dans votre arborescence de fichiers dans un nouveau dossier `TP21`. Renommez-le `NOM_bst-balance-analysis.c`
2. Écrire une fonction `bst *bst_create_random(int n_nodes)` qui génère un ABR en insérant successivement `n_nodes` valeurs aléatoires choisies entre 0 et `RAND_MAX`.
3. Écrire une fonction `void write_barchart(char *path, int n_samples, int n_nodes)` qui calcule le diagramme en bâtons des hauteurs pour `n_samples` arbres de `n_nodes` nœuds générés aléatoirement avec la fonction précédente. Cette fonction écrit les valeurs de ce diagramme en bâtons dans un fichier de chemin `path` au format CSV.
   On utilisera une virgule comme séparateur pour le format CSV. On testera que l'écriture du fichier est correcte avec de petites valeurs de `n_samples` et `n_nodes`. **Consigne : c'est le code C qui doit calculer les valeurs du diagramme en bâtons, pas le script Python qui va suivre ! !. Indication pour le calcul du diagramme : on a vu quelle était la pire (la plus grande) hauteur possible pour un arbre binaire...**
4. Écrire un script python `NOM_plot_barchart.py` qui permet de représenter le diagramme en bâtons des valeurs à partir du fichier CSV généré par votre code C. *Indication : on pourra utiliser la fonction `bar` du package Python `matplotlib.pyplot`*
5. Adapter le code C et le script Python pour que le diagramme en bâtons généré indique également la hauteur optimale (la plus faible) que l'on pourrait obtenir pour un arbre à `n_nodes` nœuds.
6. Faire des tests, analyser les diagrammes en bâtons obtenus et conclure.

Définition 2 (Arbre AVL (Addison-Velsky and Landis))

On définit inductivement la notion d'arbre binaire AVL :

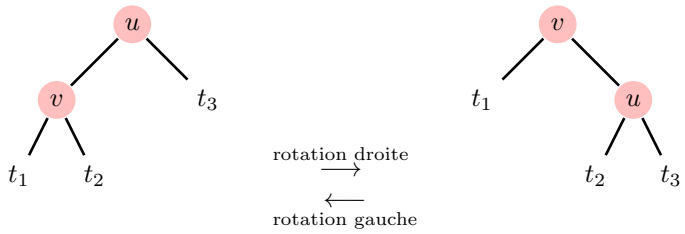
- L'arbre vide est AVL ;
- Un arbre $N(\ell, x, r)$ est AVL si :
  - ℓ est AVL
  - r est AVL
  - les hauteurs des sous-arbres ℓ et r diffèrent d'au plus 1 :

$$|h(r) - h(\ell)| \leq 1$$

Exercice 2 (Les arbres AVL sont équilibrés).

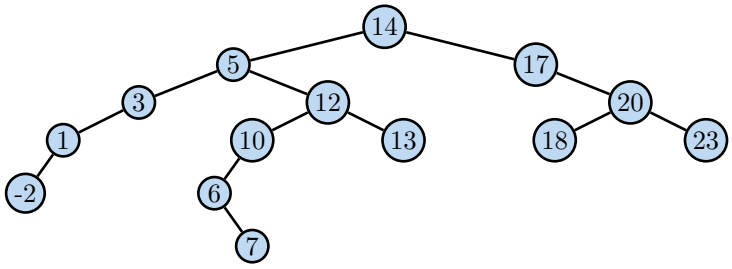
1. Soit $\mathcal{A}$ l'ensemble des arbres AVL. Montrer qu'il existe une constante $c > 1$ telle que, pour tout $t \in \mathcal{A}$, on a :
$$c^{h(t)+1} - 1 \leq n(t) \leq 2^{h(t)+1} - 1$$
Que vaut c ?
2. En déduire que les arbres AVL sont équilibrés au sens de la première définition.
3. Donner la plage de valeurs de hauteur pour des arbres AVL à 1000 nœuds puis à 100000 nœuds. Quelle aurait été la borne supérieure de cette plage de valeurs sans la propriété AVL ?

On peut donc essayer d'équilibrer nos ABR en tentant des modifications qui permettent de les rendre AVL.
Pour cela, nous allons utiliser deux opérations de modification (transformateurs) : la rotation gauche autour d'un nœud, et la rotation droite autour d'un nœud :

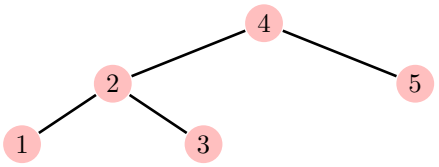


Exercice 3 (Opérations de rotation (gauche ou droite) autour d'un nœud).

1. Au considère l'ABR de la figure ci-dessous (c'est le même arbre que celui du premier TP sur les ABR !). Dessiner l'arbre obtenu après une rotation droite autour du nœud d'étiquette 10.



2. A partir de l'arbre modifié, faire à présent une rotation gauche autour du nœud d'étiquette 17 de l'arbre donné ci-dessus.
3. Faire à présent une rotation droite autour du nœud d'étiquette 14.
4. Montrer que les opérations de rotation gauche et droite conservent la propriété d'ABR (et vérifiez le sur vos manipulations précédentes !)
5. Copier le code `NOM_bst-balance-analysis.c` dans un nouveau fichier `NOM_bst-avl.c`. Commenter le `main` et créer un nouveau `main` vide.
6. Dans le fichier `NOM_bst-avl.c`, implémenter la fonction `void rotate_right(bst *n)` effectuant une rotation du nœud d'adresse `n`. **On procédera en échangeant les étiquettes des nœuds** u **et** v et en mettant à jour les liens entre les nœuds. Pensez à mettre à jour les parents ! Pensez à bien tester tous les cas possibles pour chaque rotation : pratiquez une programmation défensive et sure !
7. Tester cette fonction dans votre nouveau `main`, sur le petit cas ci-dessous en tournant autour de la racine :



8. Même chose pour `void rotate_left(bst *n)`.
9. Vérifier, en codant des tests en dur dans le `main`, que les deux transformateurs implémentés sont bien réciproques l'un de l'autre.
10. Tester sur l'exemple travaillé sur les 3 premières questions de cet exercice.

Nous allons maintenant voir comment, à partir d'un arbre légèrement déséquilibré, on peut rétablir l'équilibre par des rotations.
Pour cela, nous allons créer un champ de type entier supplémentaire à chaque nœud de l'arbre pour stocker la hauteur du sous-arbre ayant ce nœud pour racine. A partir de ces hauteurs, nous pourrons alors calculer une valeur pour chaque nœud, appelée `facteur d'équilibre` bf, définie ainsi :

Définition 3 (Facteur d'équilibre)

Le facteur d'équilibre associé à un nœud est un entier mesurant le déséquilibre entre les hauteurs de ses sous-arbres droit et gauche. Il est calculé par induction de la manière suivante :

$$\begin{cases} \mathrm{bf}(E) & = 0 \\ \mathrm{bf}(N(\ell, x, r)) & = h(r) - h(\ell) \end{cases}$$

Remarque. Ainsi, un arbre est AVL si les facteurs d'équilibrage de tous ses nœuds sont au plus égaux à 1 en valeur absolue (-1, 0 ou 1).

Exercice 4 (Propriété AVL).

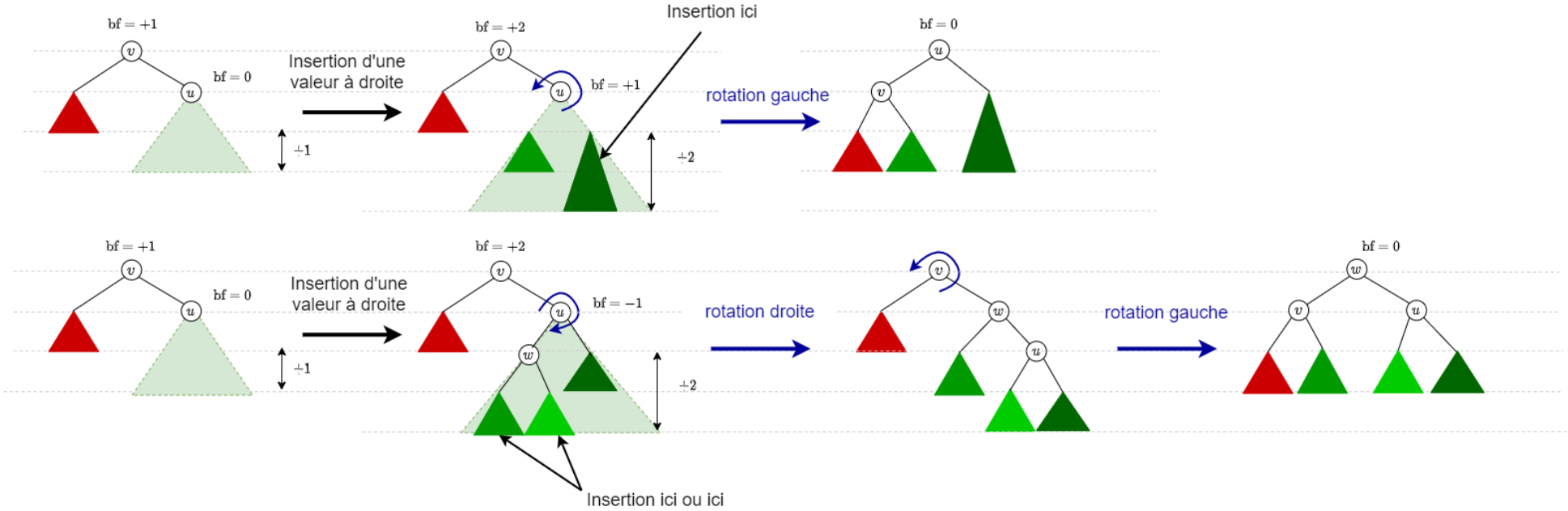
1. Dessiner un arbre AVL et un arbre non AVL avec au plus 10 nœuds.
2. Adapter le code `NOM_bst_avl.c` pour pouvoir stocker, en chaque nœud, la hauteur du sous-arbre dont ce nœud est la racine. Créer une fonction `bst_update_height` qui recalcule toutes les hauteurs de tous les nœuds. Tester et vérifier votre fonction.
3. Écrire une fonction `int balance_factor(bst *t)` qui calcule le facteur d'équilibre d'une nœud.
4. Écrire une fonction `bool bst_is_avl(bst *t)` qui indique si un arbre est, ou non, un arbre AVL. La tester sur les arbres dessinés au début de l'exercice.

On se place dans le cas où on insère une nouvelle valeur dans l'arbre. Notre objectif est de rétablir la propriété AVL après chaque nouvelle insertion. Deux cas peuvent se produire :

— La nouvelle insertion est telle que tous les facteurs d'équilibrage restent à -1, 0 ou 1 : dans ce cas, l'arbre est AVL est il n'y a rien à faire ;

— Sinon, cela signifie que l'un des sous-arbres t a un facteur d'équilibre qui est passé à $\mathrm{bf}(t) = +2$ (déséquilibre à droite) ou à $\mathrm{bf}(t) = -2$ (déséquilibre à gauche). Prenons le cas $\mathrm{bf}(t) = +2$ (insertion et création d'un déséquilibre à droite). Comme l'arbre avant insertion était supposé AVL, cela signifie que, avant insertion, $\mathrm{bf}(t)$ était au maximum égal à 1. Mais, comme une insertion dégrade au pire d'une unité, le facteur avant insertion devait être à $+1$ pour pouvoir atteindre $+2$ après insertion.

Voyons comment se forme un tel déséquilibre, par exemple en étudiant ce qui peut se passer lors de l'insertion d'un élément dans le sous-arbre droit, qui crée un déséquilibre à droite. **On explique également comment rétablir l'équilibre grâce à une ou deux rotations locales**. Le cas d'un déséquilibre à gauche est tout à fait symétrique (dessinez-le !)


bf = +1
v
u
bf = 0
Insertion d'une
valeur à droite
±1
bf = +2
bf = +1
Insertion ici
±2
rotation gauche
bf = 0
bf = +1
bf = 0
Insertion d'une
valeur à droite
±1
bf = +2
bf = −1
w
±2
rotation droite
rotation gauche
bf = 0
Insertion ici ou ici

A chaque nouvelle insertion, on vérifiera lors de la remontée récursive les facteurs d'équilibre dans tous les sous-arbres impactés par la nouvelle insertion. Si un sous-arbre est déséquilibré, c'est-à-dire si son facteur d'équilibre est tel que $|\mathrm{bf}(t)| > 1$, alors on modifie localement l'arbre avec des rotations pour tenter de rétablir l'équilibre avant d'aller regarder l'impact sur l'arbre parent.

Exercice 5 (Algorithme d'insertion avec rééquilibrage AVL).

1. Détailler toutes les étapes de l'algorithme d'insertion avec rééquilibrage lorsque l'on insère la liste de valeurs suivantes, dans cet ordre : `1, 2, 3, 4, 5, 6, 7, 8`.
2. Même question pour `8, 3, 7, 9, 11, 2, 1, 6, 4, 10`.
3. Implémenter une nouvelle fonction d'insertion `bst *bst_insert_balance(int k, bst *t)` qui insère une valeur dans un ABR à la manière habituelle puis rééquilibre l'arbre pour retrouver la propriété AVL si cela est nécessaire.
4. Tester votre fonction sur de petits exemples stimulant chaque branche d'exécution du flot de contrôle. Tester ensuite sur les exemples détaillés aux questions 1 et 2.

Exercice 6 (Comparaison expérimentale de l'équilibre d'arbres AVL et non AVL).

1. Copier le fichier `NOM_bst-avl.c` dans un nouveau fichier `NOM_bst-avl-analysis.c`.
2. En partant d'une copie de la fonction `bst_create_random` du premier exercice, créer une fonction `bst *bst_create_random_balanced(int n_nodes)` qui créer un ABR par insertion de valeurs aléatoires en utilisant la technique d'équilibrage vue plus haut.
3. Adapter la fonction `write_barchart` pour qu'il soit possible de passer en argument d'entrée un booléen `balanced` indiquant si on souhaite, ou non, utiliser une insertion avec rééquilibrage lors de la création des arbres aléatoires.
4. Commenter la fonction `main` correspondant aux tests de l'exercice précédent, et reprendre le `main` du fichier d'analyse de l'exercice 1. Modifier cette fonction `main` pour que soient écrits deux fichiers `random_bst.csv` et `random_bst_unbalanced.csv` correspondant aux deux diagrammes en bâtons pour une génération avec équilibrage et sans équilibrage, respectivement.
5. Adapter le script Python `plot_barchart.py` pour pouvoir afficher les deux diagrammes en bâtons sur la même graphique : le diagramme en bâtons sans rééquilibrage en rouge, le diagramme en bâtons avec rééquilibrage en bleu.
6. Rajouter quelques lignes dans votre script Python pour faire apparaître les deux bornes de hauteurs pour les arbres AVL sur le graphe. On pourra repartir de la formule

   $$c^{h(t)+1} - 1 \leq n(t) \leq 2^{h(t)+1} - 1$$

   et retourner voir l'exercice correspondant et sa correction.
7. Observer vos résultats sur plusieurs cas et conclure.