STPI 2A – Algorithmes et Programmation 4
TP: Arbres et récursivité

Un TP sur les arbres pas à pas. Le nombre d'exercices ne doit pas vous faire peur.

Un compte rendu au format PDF est attendu pour ce TP, en plus des sources de votre programme (dans un zip). Attention : **déposer à part le PDF et les sources du programme sur Celene**. Il vous faudra donc déposer 2 fichiers lors de votre rendu.

Le compte rendu contiendra les réponses lorsque des questions sont posées pour certains exercices ET les résultats d'exécution des challenges (ce que le programme du challenge imprime dans la console). Les challenges seront générés à l'aide d'un site web (voir annexe ⋆).

Code couleur : Très facile | Normal | Difficile → **Challenge / Question**.

Préparation Environnement de travail

Ceci est un court rappel de l'initialisation d'un projet CodeBlocks. Si nécessaire, plus de détails peuvent être obtenus dans le leaflet *Bonne pratiques* disponible sur celene. Dans le cadre de ce TP, nous allons utiliser un ensemble de fichiers source qui vont définir le projet. Ces fichiers sont logiquement associé dans le même projet et stockés dans le même répertoire (c'est plus simple).

Exercice 1 Créez un nouveau projet CodeBlocks ayant pour nom **TP-Arbre-VOTRENOM** (où **VOTRENOM** sera judicieusement instancié). Faites en sorte que le projet soit bien stocké dans un répertoire valide (Champ *Folder to create project in :* ne soit pas vide lors de la création du projet).

Les différentes étapes d'ajout d'un nouveau fichier au projet seront détaillées pour la création d'un nouveau type de fichier (.h ou .c), vous devrez le reproduire à la suite pour les autres.

Ce ne sont *a priori* que des rappels...

Exercice 2 Dans le projet CodeBlocks, ajoutez un fichier `arbre.h` au travers du menu `File/New/File`. Assurez-vous que le fichier est bien dans le même répertoire que `main.c` en utilisant le bouton `...` à droite de la boite de texte *Filename with full path :*. La boite de texte *Header guard word :* se remplit avec `ARBRE_H_INCLUDED`. N'y touchez pas ! Cette précaution est importante dans le cas où un fichier d'entête est inclus plusieurs fois, directement ou indirectement, dans un fichier `.c`. Elle évite simplement de définir plusieurs fois les structures utilisées, ce qui est interdit en C. Les définitions que l'ajoutera par la suite seront mises avant la dernière ligne `#endif`, sinon la précaution précédente est inopérante.

Exercice 3 Ajoutez de la même manière (en adaptant bien sûr) un fichier `arbre.c` dont la première ligne sera `#include "arbre.h"`. Il n'y a pas de *Header Guard word*, c'est normal !

1 Construction d'arbres et affichage simple

1.1 Construction d'un arbre à la force du poignet (45 min)

Exercice 4 Dans arbre.h, concevoir une structure *noeud* ayant 4 champs :
- cle, de type `int`
- valeur, de type `TValue`

— FGauche, pointeur vers le fils gauche de ce nœud.
— FDroit, pointeur vers le fils droit de ce nœud.

Le principe d'un nœud est de stocker la structure de l'arbre au travers des pointeurs sur les autres nœuds et l'information importante est mise dans un enregistrement particulier appelé valeur dont le type est à déterminer en fonction de l'usage. Pour différentier les éléments, on utilise une clé, ici de type entier.

Pour que cela fonctionne, il faut définir le type `TValue`. Pour ce TP, on choisira dans un premier temps un entier.

Question 1 Pour que l'on ait l'impression de manipuler un arbre quand on a un pointeur sur la racine, nous proposons de faire un `typedef struct noeud arbre;`. Ainsi, faire `arbre * a;` ou `struct noeud * a;` est strictement équivalent. Quel typedef faire afin de pouvoir écrire et compiler une déclaration telle que : `noeud * n;` équivalent à `struct noeud * n;` ?

Exercice 5 Dans le fichier **`main.c`** et dans `main()`, créez un pointeur `unArbre` sur `arbre` initialisé à `NULL`.

Exercice 6 Utilisez le fichier **`arbre.c`** pour l'implémentation des primitives de manipulation d'un arbre. Implémentez la fonction `noeud * creerNoeud(int cle)` qui alloue un nœud, met son champ `valeur` égal à `1`, met son champ `cle` à `cle` et met le pointeur `FGauche` et `Fdroit` à `NULL`. Attention, le prototype de cette fonction doit être donné dans le fichier **`arbre.h`** !

Exercice 7 Prototypez la fonction avec la syntaxe `Doxygen` suivant le mode d'emploi en annexe (cf. ⊘) et générez la documentation. Pour chaque nouvelle fonction, vous devrez désormais générer la documentation associée (c'est plus simple de le faire au fur et à mesure). Pour vous le rappeler, nous signalons ce travail par le logo : ⊘.

Exercice 8 Dans le `main()`, appelez la fonction creerNoeud avec l'entier 666 et stockez le résultat de retour dans la variable unArbre. unArbre est de type `arbre *` et la fonction renvoie un type `noeud *` et pourtant cela compile.

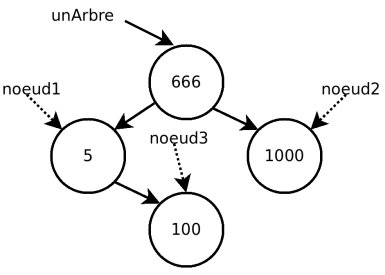
Question 2 Pourquoi obtient-on le résultat précédent ?

Exercice 9 Dans le `main()`, appelez la fonction `creerNoeud` avec l'entier 5 et stockez le résultat de retour dans une variable nommée `noeud1` de type `noeud *`.

Exercice 10 Dans **`arbre.c`**, créez une procédure `void accroche(noeud * pere, noeud * fils)` ⊘ qui accroche le nœud `fils` comme sous nœud fils de `pere` avec la règle suivante : si la clé de `pere` est supérieure à celle de `fils`, accrochez à gauche, sinon accrochez à droite. Si les deux clés sont égales, on ne fait rien (pour l'instant). N'oubliez pas de prototyper cette fonction dans **`arbre.h`**. Ajoutez l'affichage de l'information "J'accroche à gauche/à droite de %i, le noeud %i\n" (avec le côté adapté) pour tracer les accrochages. Si on ne fait pas l'accrochage, on spécifiera que le nœud est déjà rencontré par la formule suivante : "Le noeud %i est déjà présent\n"

Exercice 11 Dans le main, appelez `accroche(unArbre, noeud1);`. Vérifiez la sortie qui doit vous afficher : "J'accroche à gauche de 666, le noeud 5".

Exercice 12 Créer le nœud doublon contenant la valeur 666 et accrochez-le au nœud `unArbre`. Créez le nœud noeud2 contenant la valeur 1000, et accrochez le au nœud `unArbre`. Créez le nœud `noeud3` contenant la valeur 100, et accrochez le au nœud `noeud1`. Vous devez obtenir la sortie :




```
J'accroche a gauche de 666, le noeud 5
Le noeud 666 est deja present
J'accroche a droite de 666, le noeud 1000
J'accroche a droite de 5, le noeud 100
```

Question 3 Suivez les instructions données dans l'annexe ⋆ pour récupérer vos challenges personnalisés et les inclure à votre projet. Dans votre rapport, précisez le username utilisé pour générer

vos challenges.

Challenge 1 Dé-commentez la fonction challenge_1 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_1 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_1 dans le main.

Question 4 Dessinez l'arbre obtenu en fonction de la séquence générée. Attention, il existe un nœud racine à mettre à la racine avant tout autre ajout.

1.2 Construction automatique d'un arbre binaire de recherche (30 min)

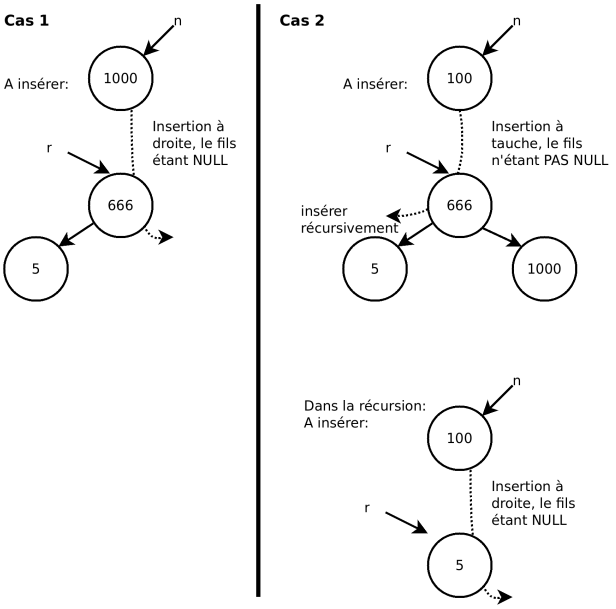
Vous avez remarqué (jetez un œil à la fonction **`challenge_1`**) que nous avons accroché "à la main" chaque nœud pour construire l'arborescence dessinée ci-avant. L'arbre construit à la main était un arbre binaire de recherche, puisqu'il respecte le critère "les nœuds à **gauche** d'un nœud ont tous une valeur de clef **plus petite strictement**" (l'arbre n'a pas de doublon). À ce stade, nous voudrions pouvoir ajouter des nœuds au niveau de la racine et qu'ils "descendent" jusqu'au bon endroit dans les feuilles.

Exercice 13 Créez une procédure ***`void`*** *`insererDansArbre (arbre * r, noeud * n)`* ⊘ dont le but est d'insérer dans l'arbre binaire de recherche (*r* n'est jamais *`NULL`*). Pour ce faire, le principe général de l'algorithme **récursif** est le suivant :

- Déterminez si l'insertion peut se faire (sans créer de doublon). Le cas échéant, voir si elle doit avoir lieu dans le fils gauche ou dans le fils droit : créez selon le cas un pointeur temporaire *`noeud * fils ;`* qui pointe vers le fils choisi.
- Examinez ce fils :

Cas 1 : s'il n'existe pas, appelez votre fonction ***`void`*** *`accroche(noeud * r, noeud * n )`* pour réaliser l'accrochage de votre nœud *n* sur le père du fils, c'est-à-dire sur *r* ;

Cas 2 : Si le fils existe, faites un appel récursif à *`insererDansArbre`* en lui passant comme premier paramètre le nœud *`fils`*. En cas de doublon, cet appel (ou les récursions suivantes) permettront de le déterminer.



Exercice 14 Mettez en commentaire la création et l'accrochage de *`noeud1`*, *`noeud2`*, *`noeud3`* dans **`main.c`**. Remplacez-les par le code ci-dessous, qui utilise votre fonction **`insererDansArbre`**. Vérifiez que la sortie console est identique à la précédente (comme celle montrée à l'exercice 12).

```
1  insererDansArbre(unArbre, creerNoeud(5));
2  insererDansArbre(unArbre, creerNoeud(666));
3  insererDansArbre(unArbre, creerNoeud(1000));
4  insererDansArbre(unArbre, creerNoeud(100));
5  insererDansArbre(unArbre, creerNoeud(5));
6  insererDansArbre(unArbre, creerNoeud(100));
```

Challenge 2 Dé-commentez la fonction challenge_2 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_2 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_2 dans le main.

Exercice 15 Créez une fonction `void insererTableau(arbre * r, int * tableau, int taille )` ⊘ qui permet de créer et d'insérer tous les nœuds du tableau dans l'arbre `r`. Dans votre main créez un tableau d'entiers contenant [1, 45, 800, -40, 300, 4, 5, 333, 1001, 700] et insérez tous ces nœuds, en plus des nœuds 666, 5, 1000, 100 déjà présents. Vous devez obtenir, en sortie quelque chose du genre :

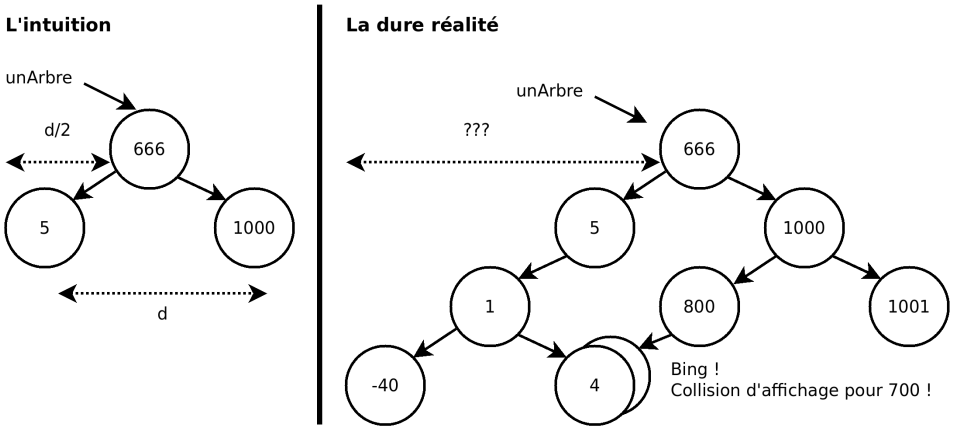
```
J'accroche a gauche de 666 le noeud 5
Le noeud 666 est deja present
J'accroche a droite de 666 le noeud 1000
J'accroche a droite de 5 le noeud 100
Le noeud 5 est deja present
Le noeud 100 est deja present
J'accroche a gauche de 5 le noeud 1
J'accroche a gauche de 100 le noeud 45
J'accroche a gauche de 1000 le noeud 800
J'accroche a gauche de 1 le noeud -40
J'accroche a droite de 100 le noeud 300
J'accroche a droite de 1 le noeud 4
Le noeud 5 est deja present
J'accroche a droite de 300 le noeud 333
J'accroche a droite de 1000 le noeud 1001
J'accroche a gauche de 800 le noeud 700
```

Challenge 3 Dé-commentez la fonction challenge_3 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_3 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_3 dans le main.

Exercice 16 A partir de cette étape, si votre challenge semble réussi, désactivez les printf qui imprime "J'accroche bla bla bla" dans votre fonction **`accroche`**.

1.3 Affichage d'un arbre : pas si simple ! (60 min)

Dans cette section, on se propose d'afficher un arbre dans la console. Il s'agit de réaliser un affichage ligne par ligne. Pour vous enlever tout espoir qu'il pourrait s'agir d'un petit exercice tranquille et reposant pour l'esprit, voici un petit dessin illustratif :



Pour simplifier, on propose d'afficher de la manière suivante : la première ligne comportera la racine et tous ses fils droits. Puis, en dessous, on trouvera éventuellement des traits (| ou *) puis les fils gauche. Par rapport au dessin précédent, la sortie attendue est de la forme :

Listing 1 – Algorithme d'affichage

```
1  void afficher(arbre * r, int decalage)
2  {
3      SI r est Vide ALORS
4        retourner;
5      FIN SI
6
7      printf("%-6i", r->cle); // Affichage du noeud
8
9      SI r->FDroit existe ALORS // Affichage du noeud a droite
10       afficher(r->FDroit, decalage+6) // recurDroite: appel recursif
11     FIN SI
12
13     // Affichage du noeud a gauche
14     SI r->FGauche existe ALORS
15       printf("\n");
16       POUR i de 0 a decalage-1 FAIRE
17         Afficher un espace
18       FIN POUR
19       printf("|\n");
20       POUR i de 0 a decalage-1 FAIRE
21         Afficher un espace
22       FIN POUR
23       afficher(r->FGauche, decalage); // recurGauche: appel recursif
24     FIN SI
25 }
```

```
666---1000
|
5
```

```
666---1000--1001
      |
      800
      |
      700
|
5
|
1-----4
|
-40
```

Exercice 17 Pour réaliser cet affichage, implémentez l'algorithme donné au listing 1 dans la fonction ***void*** *afficher(arbre * r,* ***int*** *decalage)* ⊘. **Attention**, la fonction s'appelle `afficher`. Elle est récursive et différente de l'instruction `Afficher un espace` qui se traduit par un simple `printf`. Le principe est le suivant :

- J'affiche le nœud courant [1] (sans décalage).
- **recurDroite** : J'affiche le fils droit récursivement car il doit être affiché sur la même ligne de texte ; j'augmente le décalage de 6 car tous les autres affichages récursifs doivent suivre ce décalage.
- **recurGauche** : Enfin, j'affiche le fils gauche récursivement avec le décalage éventuel.

On peut essayer de suivre cet algorithme sur le dessin de la façon suivante : on appelle *`afficher(unArbre,0);`* car le nœud 666 doit avoir un décalage de 0. Puis, à cause de **recurDroite** (Ligne 10) on part dans l'affichage du nœud 1000, avec décalage de 6. Récursivement, on affiche le nœud

1. Le code de format %-6i permet d'afficher un entier et de compléter avec des espaces afin qu'il y ait toujours 6 caractères affichés. Autrement dit, "666" s'affiche en tant que "6 6 6 _ _ _".

1000. Puis, à cause de **recurDroite** (Ligne 10) on part dans l'affichage du nœud 1001. Le nœud 1001 n'ayant aucun fils, on revient dans la récursion qui concerne le nœud 1000. Pour ce nœud 1000, on continue à exécuter la suite Ligne 14 : on va afficher son fils gauche (800). Pour cela, on affiche autant d'espace que le décalage, puis le caractère "|\n". Puis on affiche à nouveau autant d'espace que le décalage et on lance l'affichage récursif sur le fils gauche (800), en ligne 23. On obtient donc les 3 lignes du listing 2. Etc.

Listing 2 – Affichage résultant (exo 17)

```
666   1000  1001
      |
      800
      |
      700
|
5     100   300   333
      |
      45
|
1     4
|
-40
```

Listing 3 – Affichage résultant (exo 18)

```
666---1000--1001
      |
      800
      |
      700
|
5-----100---300---333
      |
      45
|
1-----4
|
-40
```

Exercice 18 Optionnel. Cosmétique : remplacez les espaces entre les nœuds d'une même ligne comme il était prévu au début de cette section [2].

Challenge 4 Dé-commentez la fonction challenge_4 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_4 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_4 dans le main.

Dans la suite, on aimerait pouvoir afficher n'importe quelle valeur calculée en un sommet de l'arbre. En particulier, nous avons créé un attribut valeur dans la structure noeud. Pour arriver à nos fins, nous allons utiliser les pointeurs sur fonctions qui permettent de passer en argument d'une fonction une autre fonction, pourvu que cette dernière ait le bon prototype. Nous allons commencer par créer deux fonctions de test qui seront affichées par la suite.

Exercice 19 Créez une fonction ***int*** *getCle(arbre * r)* qui retourne simplement la valeur de l'attribut `cle` de la racine de l'arbre r. Si celui-ci n'existe pas, il retourne 0.

Exercice 20 Créez sur le même principe une fonction ***int*** *getValeur(arbre * r)* qui retourne la valeur de la racine r.

Exercice 21 Créez la fonction **afficherFonction** dont le prototype est donné par le listing 4. Le principe global consiste à reprendre la structure de la fonction **afficher**. Pour passer une fonction en paramètre, il suffit de donner son nom comme dans le listing 5.

Listing 4 – prototype de la fonction `afficherFonction`

```
1 /** \brief Affiche l'arbre r avec en chaque sommet le resultat de la fonction mafct
2  *
3  * \param r arbre*: arbre a parcourir
4  * \param decalage int: decalage a partir duquel on affiche l'arbre
5  * \param mafct int (*mafct)(arbre * r): la fonction a imprimer. Celle-ci prend un
       arbre en parametre et retourne un entier
6  * \return void
7  *
8  */
```

2. Si vous avez besoin du log, compilez avec l'option -lm.

```
9 void afficherFonction(arbre * r, int decalage, int (*mafct)(arbre * r));
```

Listing 5 – appel de la fonction `afficherFonction`

```
1 afficherFonction(unArbre, 0, getCle);
2 printf("\n");
3 afficherFonction(unArbre, 0, getValeur);
```

Exercice 22 Le résultat de la première ligne donne l'affichage de l'arbre tel que vous l'avez obtenu avec l'appel de `afficher`. Celui de la troisième ligne doit donner la même structure d'arbre avec des 0 partout. Modifiez la fonction `creerNoeud` pour que le résultat de l'affichage de cette dernière ligne soit donnée par le listing 6.

Listing 6 – Affichage résultant (exo 22)

```
1-----1-----1
      |
      1
      |
      1
|
1-----1-----1-----1
      |
      1
|
1-----1
|
1
```

Challenge 5 Dé-commentez la fonction challenge_5 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_5 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_5 dans le main.

2 Algorithmique sur les arbres

2.1 Recherche de nœuds(20 min)

Question 5 On souhaite retrouver un nœud dans l'arbre, à partir d'un entier *v*, et renvoyer l'adresse de ce nœud s'il existe. Que doit-on renvoyer si ce nœud n'existe pas ? Quelle est la signature de cette fonction *rechercher* ?

Exercice 23 Implémentez la fonction *rechercher* ⊘. Testez-là dans le main à l'aide du code suivant qu'il faut compléter au niveau des commentaires :

```
1 noeud * cherche100 = .... // appel a rechercher pour chercher le noeud 100
2 if (cherche100 != NULL)
3   printf("J'ai trouve le noeud 100.\n");
4 noeud * cherche101 = .... // appel a rechercher pour chercher le noeud 101
5 if (cherche101 != NULL)
6   printf("J'ai trouve le noeud 101.\n");
```

qui doit donner :

```
J'ai trouve le noeud 100.
```

Exercice 24 Implémentez la fonction
`noeud * rechercherDerniereLettre(arbre * unArbre);` ⊘ qui recherche le nœud correspondant à la dernière lettre de votre login (en capitale). On pourra remarquer qu'un caractère (type **`char`**) est un sous-type d'entier et que l'on peut directement écrire la déclaration suivante : `int var = 'A';`, ce qui aura pour effet d'initialiser la variable `var` par la position du caractère `A` dans la table ASCII, en l'occurrence 65.

Challenge 6 Dé-commentez la fonction challenge_6 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_6 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_6 dans le main.

2.2 Prise en compte du nombre d'occurrences (20 min)

Dans cette partie, on cherche à utiliser l'attribut `valeur` du nœud qu'on avait oublié. On va utiliser cet attribut pour compter le nombre de fois que l'on essaie d'insérer une clé spécifique (au lieu de ne rien faire actuellement).

Exercice 25 Modifiez les fonctions **`creerNoeud`** et **`accroche`** afin de comptabiliser le nombre de fois où une clé a été insérée dans l'arbre. L'affichage obtenu par la fonction `afficherFonction` avec la fonction `getValeur` doit alors être

```
2-----1-----1
      |
      1
      |
      1
|
3-----2-----1-----1
      |
      1
|
1-----1
|
1
```

Exercice 26 Implémentez la fonction `int nbVal(arbre * unArbre, int uneCle)` qui retourne le nombre de fois que la clé `uneCle` a été insérée dans l'arbre.

Challenge 7 Dé-commentez la fonction challenge_7 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_7 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_7 dans le main.

Question 6 On suppose que l'on n'insère que des nœuds isolés (feuilles) dans l'arbre. Montrer que la gestion des doublons induit une perte de mémoire.

Exercice 27 Modifiez la fonction **`insererDansArbre`** pour éviter les fuites mémoire.

2.3 Déforestation (20 min)

Exercice 28 Déforestation sauvage : on souhaite couper tous les fils d'un nœud. Écrire la fonction `void deforestationSauvage(noeud * n);` ⊘ qui supprime les deux fils de ce nœud *n*. Testez son efficacité en *déforestant* le nœud 100 et en affichant votre arbre à nouveau.

Listing 7 – Affichage résultant de la déforestation sauvage du noeud 100

```
666---1000--1001
```

```
      |
      800
      |
      700
|
5-----100
|
1-----4
|
-40
```

Challenge 8 Dé-commentez la fonction challenge_8 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_8 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_8 dans le main.

Exercice 29 Déforestation plus "écolo" : au lieu d'avoir perdu tous vos nœuds de l'arbre lors de la découpe, vous auriez pu les collecter (pour en faire des meubles). Autrement dit, tous vos nœuds fils du nœud 100 auraient dû avoir droit à un coup de `free`. Réalisez une fonction récursive `void deforestation(noeud * n)` ⊘ qui réalise les opérations suivantes :

- Si le paramètre *n* est NULL, sortir ;
- J'appelle récursivement la déforestation du fils gauche ;
- Je détruis le fils gauche (free) ;
- J'appelle récursivement la déforestation du fils droit ;
- Je détruis le fils droit (free) ;
- Je mets les pointeurs de mes fils à NULL.

Exercice 30 Mettez en commentaire l'appel à `deforestationSauvage` dans le main et appelez `deforestation(cherche100)` ⊘. Vous devez obtenir le même résultat que le Listing 7 mais avec le sentiment d'avoir bien nettoyé la mémoire.

Exercice 31 Pour se conforter dans l'idée du travail bien fait, complétez la fonction `deforestation` pour qu'elle affiche à chaque fois qu'elle libère un nœud le message suivant : "Déforestation : élimination du noeud %i fils gauche/fils droit de %i\n". On doit obtenir le résultat avant l'affichage de l'arbre.

```
Deforestation : elimination du noeud 45 fils gauche de 100
Deforestation : elimination du noeud 333 fils droit de 300
Deforestation : elimination du noeud 300 fils droit de 100
```

Challenge 9 Dé-commentez la fonction challenge_9 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_9 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_9 dans le main.

Question 7 Pourquoi peut-on appeler plusieurs fois `deforestation(cherche100);` dans le main sans qu'il n'y ait de plantage à cause d'un appel à `free` impossible (à cause d'une désallocation déjà faite) ?

Dans cette section, on s'intéresse à des algorithmes un peu plus avancés qui nécessitent des combinaisons de parcours, découpe, ou raccrochage.

2.4 Parcours et coupes (45 min)

Exercice 32 Générez un nouvel arbre de racine 666 et ajouter le tableau de nœuds [1000,5,100,1,6,200]. Affichez-le :

```
666---1000
|
5-----100---200
      |
      6
|
1
```

Exercice 33 Faites une fonction *int somme(arbre * r)* ⊘ qui calcule la somme de l'arbre. Vous devez obtenir une somme de 1978 sur cet arbre.

Exercice 34 Réalisez une fonction *void parcoursProfondeur(arbre * r)* ⊘ qui écrit sur une ligne de texte, le parcours en profondeur d'un arbre. L'algorithme est bien sûr récursif, et donne pour notre arbre :

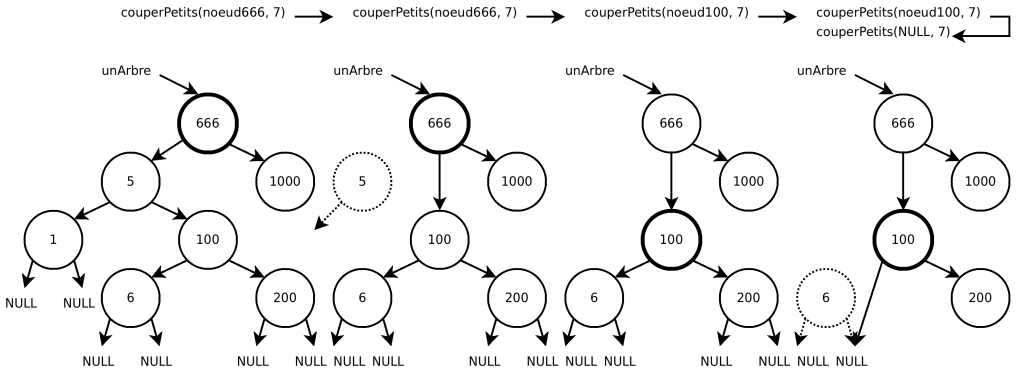
```
666 5 1 100 6 200 1000
```

Challenge 10 Dé-commentez la fonction challenge_10 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_10 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_10 dans le main.

Exercice 35 Soit **seuil** une valeur **toujours inférieure à la valeur de la racine** de l'arbre. Écrire une fonction *void couperPetits(arbre * a, int seuil)* ⊘ qui coupe les nœuds dont la valeur sont inférieurs à *seuil* (élagage). Utilisez la valeur **seuil=8** pour vos premiers tests. Quelques conseils pour vous aider dans la réflexion sur cet algorithme pas si trivial :

- on ne peut pas utiliser *deforestation(noeud * n)* (dommage !)
- comme le seuil est plus petit que la valeur de la racine, il faut :
  - **regarder son fils gauche**.
  - **s'il est plus petit strictement (il doit être coupé), raccorder la racine au fils droit du fils gauche, et faire une récursion à gauche**. Par exemple sur le dessin pour le premier appel *couperPetits(noeud666,7)*, 5 est *plus petit* que 7 : on coupe et on raccorde son fils droit 100 à 666.
  - **s'il est plus grand ou égal à seuil, alors faire une récursion à gauche**. Par exemple, sur le dessin pour le second appel *couperPetits(noeud666,7)*, 100 est *plus grand* que 7 : il doit être conservé et on fait juste une récursion à gauche sur le nœud 6.

Pour bien réussir cet exercice, il n'est pas inutile de faire des schémas et de tester plusieurs valeurs de seuil (2, 6, 8, 150, 300).



Challenge 11 Dé-commentez la fonction challenge_11 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_11 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_11 dans le main.

Exercice 36 **Optionnel mais pas trop dur**. Mettre en évidence dans l'élagage les nœuds qui ont éliminés et faire en sorte que l'opération ne soit pas trop sauvage. On peu avoir le résultat suivant :

```
Elimination de ce qui etait dans le FG (6)
Elimination de l'ancien FG (6) et liberation de la memoire
Elimination de ce qui etait dans le FG (5)
Deforestation : elimination du noeud 1 fils gauche de 5
Elimination de l'ancien FG (5) et liberation de la memoire
666~~~1000
|
100~~~200
```

Exercice 37 **Optionnel et difficile (pour ceux qui auront tout fini !)**. Faire un élagage qui peut éventuellement supprimer la racine. Tester avec 800 et 1200.

Exercice 38 Pour la suite, désactivez dans le main l'appel à la fonction *couperPetits* afin de retrouver l'arbre de départ de cette section et les affichages spécifiques dans *deforestation* et si nécessaire dans *couperPetit*.

3 Vers des arbres équilibrés

La construction d'un ABR peut produire un arbre totalement déséquilibré. Pour s'en rendre compte, il suffit d'insérer successivement les valeurs $1, 2, 3, \ldots$. On crée ainsi un arbre qui ressemble plus à une liste chaînée. Sans aller vers de telles extrémités, l'arbre obtenu en général n'est pas très compact. La méthode AVL permet d'obtenir des arbres binaires qui se rapprochent des arbres binaires équilibrés et dont la hauteur est logarithmique en fonction du nombre de sommets.

3.1 Évaluation du déséquilibre

Exercice 39 Faites une fonction ***int*** *initHauteur(arbre * a)* qui initialise la variable **valeur** pour tous les sommets de l'arbre par la hauteur de l'arbre enraciné en chaque sommet. L'algorithme utilisé est clairement récursif.

Exercice 40 Affichez le résultat.

Exercice 41 Faites une fonction ***int*** *difference(arbre *a)* qui calcule la différence des hauteurs des fils droit et gauche du sommet de l'arbre. On suppose que l'attribut `valeur` a bien été initialisé au préalable. On rappelle que la hauteur d'un arbre vide vaut -1. Affichez le résultat avec **`afficherFonction`**. On obtient alors sur l'arbre initialement produit. On constate que l'arbre est bien équilibré car toutes les différences sont comprises entre -1 et 1. (Afin de bien voir les nombres négatifs, on peut faire évoluer un peu l'affichage des lignes horizontales)

```
1~~~~~1~~~~~0
      |
      1
      |
      0
|
-1~~~~-1~~~~-1~~~~0
      |
      0
|
0~~~~~0
|
0
```

Exercice 42 Testez à partir d'un nouvel arbre $r2$ où on insère successivement à la racine positionnée à 100 les valeurs [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 110, 120, 130, 140, 150, 160].

```
Arbre
100~~~110~~~120~~~130~~~140~~~150~~~160
|
0~~~~~10~~~~20~~~~30~~~~40~~~~50~~~~60~~~~70~~~~80~~~~90
Hauteurs dans l'arbre
10~~~~5~~~~~4~~~~~3~~~~~2~~~~~1~~~~~0
|
9~~~~~8~~~~~7~~~~~6~~~~~5~~~~~4~~~~~3~~~~~2~~~~~1~~~~~0
Differences dans l'arbre
4~~~~~-5~~~~-4~~~~-3~~~~-2~~~~-1~~~~0
|
-9~~~~-8~~~~-7~~~~-6~~~~-5~~~~-4~~~~-3~~~~-2~~~~-1~~~~0
```

Exercice 43 Faites une fonction `int estAVL(arbre * r)` qui indique si un arbre est un AVL. On suppose que l'attribut `valeur` est initialisé avec les hauteurs.

```
1 if (estAVL(unArbre))
2   printf("Cet arbre est un AVL\n");
3 else
4  printf("Cet arbre n'est pas un AVL\n");
5 if (estAVL(r2))
6   printf("Cet arbre est un AVL\n");
7 else
8  printf("Cet arbre n'est pas un AVL\n");
```

qui doit donner :

```
Cet arbre est un AVL
Cet arbre n'est pas un AVL
```

Challenge 12 Dé-commentez la fonction challenge_12 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_12 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_12 dans le main.

3.2 Quelques outils pour faire automatiquement un AVL

Deux opérations fondamentales permettent d'équilibrer un arbre binaire de recherche. Ce sont les opérations de rotation à droite et à gauche d'un arbre. Ces opérations sont décrites informellement dans la figure 1. Sur le schéma, les zones jaunes, vertes et roses correspondent à des sous-arbres, c'est-à-dire, formellement des arbres. Les sommets orange (x) et bleu (y) sont les pivots pour la rotation. Dans le cas de la rotation droite, l'idée est de faire remonter le fils gauche de x dans l'arbre et donc de diminuer la hauteur du fils gauche du sommet de l'arbre. Si ce fils a une hauteur h, dans l'arbre de gauche, son impact pour le fils gauche de la racine de l'arbre est $h + 1$, tandis que son impact sur l'arbre de droite est seulement h. En reprenant cet argument pour les deux autres sous-arbres (vert et rose), on voir que l'on peut réduire certaines différences de hauteur et se rapprocher d'un AVL. Ces opérations préservent la propriété d'être un ABR. L'exemple suffit pour s'en persuader, mais une preuve formelle est simple à établir.

Question 8 Donner le prototype en pseudo-code de la fonction **`Rotation-Droite`**. En particulier, précisez le passage de paramètre. En déduire le prototype C de cette fonction.

Exercice 44 Faites une fonction ***`void`*** `rotationDroite(arbre ** pr)` qui réalise l'opération décrite sur la figure 1. Si l'arbre ne possède pas de fils gauche, la fonction ne fait rien.

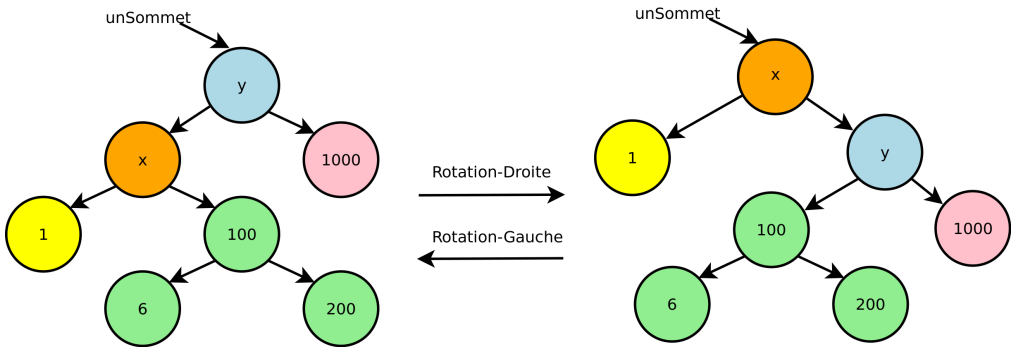


FIGURE 1 – Rotation sur un ABR

```
1 rotationDroite(&unArbre);
2 printf("Apres Rotation Droite :\n");
3 afficher(unArbre, 0);
4 initHauteur(unArbre);
5 printf("Differences dans l'arbre\n");
6 afficherFonction(unArbre, 0, difference);
```

qui doit donner :

```
Apres Rotation Droite :
5~~~~~666~~~1000~~1001
            |
            800
            |
            700
      |
      100~~~300~~~333
      |
      45
|
1~~~~~4
|
-40
```

```
Differences dans l'arbre
-2~~~~0~~~~~1~~~~~0
            |
            1
            |
            0
      |
      -1~~~~-1~~~~0
      |
      0
|
0~~~~~0
|
0
```

Exercice 45 Afin de pouvoir rééquilibrer notre arbre à qui on a fait beaucoup de mal, écrivez la fonction analogue **`rotationGauche`**.

Challenge 13 Dé-commentez la fonction challenge_13 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_13 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_13 dans le main.

3.3 Rééquilibrage d'un ABR en AVL

Ici, cela se complique vraiment, du moins quand on cherche à coder ce que l'on cherche à faire. D'un point de vue algorithmique, l'idée est très simple. Tant que notre arbre n'est pas encore un AVL (ça, on sait faire), on cherche un sommet déséquilibré et on applique la bonne rotation pour chercher à équilibrer l'arbre. Cette stratégie simple ne marche pas toujours car le choix des sommets sur lesquels faire la rotation est important. Ce qui parait le plus simple n'est pas ce qui fonctionne. En effet, si on prend un sommet proche de la racine, on ne touche pas l'arbre en profondeur et

l'algorithme boucle à l'infini. La première subtilité consiste à chercher un sommet déséquilibré le plus proche possible des feuilles.

Chercher un sommet déséquilibré, c'est assez facile : c'est une recherche simple. Même si ce n'est pas suffisant pour la suite, cela nous donnera des idées.

Exercice 46 Faites une fonction *arbre * getSousArbreDesequilibre(arbre * r)* qui recherche un sous-arbre r' de r tel que tout sous-arbre strict de r' est équilibré. Si r est déjà équilibré, cette fonction renvoie *NULL*. Pour faire cela, on fera une recherche postfixe. L'algorithme est donc récursif. Tout d'abord, on traite les cas faciles : si l'arbre est vide ou est réduit à une feuille, il faut retourner *NULL*, car ce sont trivialement des AVL. Ensuite, on recherche d'abord si le fils (droit ou gauche) possède de manière récursive un sous-arbre déséquilibré. S'il en possède un c'est celui qui sera choisi globalement. S'il n'en possède pas, on regarde l'autre fils. Si là encore, il n'y en a pas, on regarde si la racine est déséquilibrée. Si c'est le cas, on retourne le sommet, sinon, c'est *NULL* encore une fois.

```
1  arbre * r3 = getSousArbreDesequilibre(r2);
2  printf("Apres recherche du desequilibre\n");
3  afficher(r3, 0);
4  printf("\n");
5  afficherFonction(r3, 0, difference);
6  printf("\n");
7  if(estAVL(r3))
8      printf("Cet arbre est un AVL\n");
9  else
10     printf("Cet arbre n'est pas un AVL\n");
```

qui doit donner :

```
Apres recherche du desequilibre
70~~~~80~~~~90
-2~~~~-1~~~~0
Cet arbre n'est pas un AVL
```

Cette fonction n'est malheureusement pas suffisante car, souvenez-vous, pour faire une rotation, gauche ou droite, il faut savoir où modifier les choses. Donc, connaître le sommet à équilibrer n'est pas suffisant, il faut aussi savoir dans quel sommet est stocké ce sommet. En clair, il faut savoir qui est le père (dans l'arbre) du sommet déséquilibré et savoir si c'est le fils droit ou le fils gauche. Pour trouver ce sommet, on fait une nouvelle fonction, mais dont le principe général est celui de la fonction **getSousArbreDesequilibre**.

Exercice 47 Faites une fonction *arbre * aUnFilsDesequilibre(arbre * r, **int** * pADroite)* qui retourne le sous-arbre qui a un fils déséquilibré. Par ailleurs, le paramètre *pADroite* correspond à un booléen **ADroite** passé en **OUT** qui indique si le fils trouvé est à droite ou non. En ce qui concerne l'algorithme utilisé, c'est exactement le même que celui utilisé à la question précédente, en prenant le soin de voir de quel côté du sommet il y a un problème.

Il suffit maintenant d'assembler le tout pour rééquilibrer un ABR pour le transformer en AVL.

Exercice 48 Faites une fonction *__void__ tranformerVersAVL(arbre ** pr)* qui permettra de rééquilibrer un ABR. On suppose que les hauteurs ne sont pas forcément initialisées et qu'il faut le faire en premier. L'algorithme est donné à la figure 2. Cet algorithme met en place la double rotation quand cela est nécessaire. La difficulté de cette écriture de programme est de bien comprendre les passages de paramètre, les pointeurs et le déréférencement. Bon courage !

Une fois que cela fonctionne, testez votre programme.

```
1      printf("Avant la transformation de r2 :\n");
2      afficher(r2, 0);
3      printf("\n");
4      tranformerVersAVL(&r2);
```

```
5     printf("Apres la transformation de r2 :\n");
6     afficher(r2, 0);
7     printf("\nDesequilibre :\n");
8     afficherFonction(r2, 0, difference);
9     printf("\n");
```

qui doit donner :

```
Avant la transformation de r2 :


100~~~110~~~120~~~130~~~140~~~150~~~160
|
0~~~~~10~~~~20~~~~30~~~~40~~~~50~~~~60~~~~70~~~~80~~~~90



Etape 0 : rotation Gauche en 70
Etape 1 : rotation Gauche en 60
Rotation droite prealable sur Fils droit : 80
Etape 2 : rotation Gauche en 50
Etape 3 : rotation Gauche en 40
Etape 4 : rotation Gauche en 30
Etape 5 : rotation Gauche en 30
Etape 6 : rotation Gauche en 20
Etape 7 : rotation Gauche en 20
Rotation droite prealable sur Fils droit : 60
Etape 8 : rotation Gauche en 10
Etape 9 : rotation Gauche en 10
Etape 10 : rotation Gauche en 140
Etape 11 : rotation Gauche en 130
Rotation droite prealable sur Fils droit : 150
Etape 12 : rotation Gauche en 120
Etape 13 : rotation Gauche en 0
Etape 14 : rotation Gauche en 0
Etape 15 : rotation Gauche en 110
```

```
Apres la transformation de r2 :
100~~~130~~~150~~~160
            |
            140
      |
      110~~~120
|
40~~~~60~~~~80~~~~90
            |
            70
      |
      50
|
20~~~~30
|
0~~~~~10
Desequilibre :
1~~~~~0~~~~~0~~~~~0
            |
            0
      |
      -1~~~~0
|
0~~~~~-1~~~~0~~~~~0
            |
            0
      |
      0
|
1~~~~~0
|
-1~~~~0
```

Challenge 14 Dé-commentez la fonction challenge_14 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_14 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_14 dans le main.

3.4 Maintenir un AVL lors de l'insertion

En utilisant l'algorithme précédent, on voit que l'on arrive à équilibrer un ABR totalement distordu pour en faire un AVL. Cependant, la complexité globale de ce rééquilibrage n'est pas extraordinaire. En effet, à chaque étape, on refait un calcul de hauteur, ce qui prend $O(n)$ étapes (si n est le nombre de sommets), toutes les recherches et vérifications sont aussi linéaires ($O(n)$). Par contre, les rotations se font en temps constant !

Pour comprendre ce que l'on doit faire, on va déséquilibrer notre arbre $r2$ bien équilibré en ajoutant la valeur 125. L'insertion classique place ce nouveau sommet comme fils droit de 120,

```
Fonction transformerVersAVL(In-Out Variable r : arbre) : Vide
    Variable numetape : Entier
    numetape ← 0
    initHauteur(r)
    Tant Que Non estAVL(r)
        Variable aDroite : Booléen
        Variable valeurDesequilibre : Entier
        Variable deseq : Arbre // sera de type arbre *
        Variable ouEquilibrer : Arbre // sera de type arbre **

        // On cherche d'abord un sommet à équilibrer
        deseq ← aUnFilsDesequilibre(r, aDroite)
        Si estVide(deseq) // on testera simplement s'il vaut NULL Alors
            ouEquilibrer ← r
            valeurDuDesequilibre ← difference(r)
        Sinon Si aDroite Alors
            ouEquilibrer ← deseq.Fdroit // Attention, on prend l'adresse
            valeurDuDesequilibre ← difference(deseq.Fdroit)
        Sinon
            ouEquilibrer ← deseq.Fgauche
            valeurDuDesequilibre ← difference(deseq.Fgauche)

        // Ensuite, on fait la rotation adéquate, en faisant attention a la double
        Si valeurDuDesequilibre>0 Alors
            Si difference(ouEquilibrer.Fgauche)<0 Alors
                Afficher("Rotation prealable sur le fils gauche : ", ouEquilibrer.Fgauche.cle)
                rotationGauche(ouEquilibrer.Fgauche) // Attention a la traduction en C
            Afficher("Etape ", numetape, " : rotation Droite en ", ouEquilibrer.Cle)
            rotationDroite(ouEquilibrer)
        Sinon
            Si difference(ouEquilibrer.Fdroit)>0 Alors
                Afficher("Rotation prealable sur le fils droit : ", ouEquilibrer.Fdroit.cle)
                rotationDroite(ouEquilibrer.Fdroit) // Attention a la traduction en C
            Afficher("Etape ", numetape, " : rotation Gauche en ", ouEquilibrer.Cle)
            rotationGauche(ouEquilibrer)
        initHauteur(r)
        numetape ← numetape +1
FinFonction
```

FIGURE 2 – Algorithme de rééquilibrage

créant un déséquilibre en 110. On s'aperçoit qu'il suffit de rééquilibrer ce sommet pour que cela fonctionne.

```
1   printf("Insertion d'un nouveau sommet dans un arbre equilibre\n");
2   insererDansArbre(r2, creerNoeud(125));
3   afficher(r2, 0);
4   printf("\n");
5   initHauteur(r2);
6   afficherFonction(r2, 0, difference);
7   printf("\n");
8   tranformerVersAVL(&r2);
9   printf("Apres la deuxieme transformation de r2\n");
10  afficher(r2, 0);
11  printf("\n");
12  afficherFonction(r2, 0, difference);
13  printf("\n");
```

va donner :

```
Nouveau sommet dans un arbre equilibre



100~~~130~~~150~~~160
            |
            140
      |
      110~~~120~~~125
|
40~~~~60~~~~80~~~~90
            |
            70
      |
      50
|
20~~~~30
|
0~~~~~10


0~~~~~1~~~~~0~~~~~0
            |
            0
      |
      -2~~~~-1~~~~0
|
0~~~~~-1~~~~0~~~~~0
            |
            0
      |
      0
|
1~~~~~0
|
-1~~~~0
```

```
Etape 0 : rotation Gauche en 110
Apres la deuxieme transformation de r2
100~~~130~~~150~~~160
            |
            140
      |
      120~~~125
      |
      110
|
40~~~~60~~~~80~~~~90
            |
            70
      |
      50
|
20~~~~30
|
0~~~~~10
1~~~~~0~~~~~0~~~~~0
            |
            0
      |
      0~~~~~0
      |
      0
|
0~~~~~-1~~~~0~~~~~0
            |
            0
      |
      0
|
1~~~~~0
|
-1~~~~0
```

De manière plus générale, il faut rééquilibrer seulement des sommets internes sur le chemin entre la racine et la feuille. D'un point de vue complexité, comme l'arbre est de profondeur $O(\lg n)$ et que chaque rotation prend un temps constant, l'insertion dans un AVL (qui préserve la propriété) prend aussi un temps logarithmique. Globalement, c'est donc plus efficace que l'insertion dans un ABR simple où si cela se passe mal, l'insertion peut se faire en $O(n)$. C'est le cas si on insère successivement les valeurs 1, 2, …, n.

Pour mettre tout cela en place, il faut reprendre la structure générale depuis la création jusqu'à l'insertion, cela, spécifiquement pour les AVL.

Exercice 49 Créez une nouvelle fonction `noeud * creerNoeudAVL(int cle)`. La différence avec la fonction initiale est simplement d'initialiser le champ `valeur` à 0, c'est-à-dire, à la hauteur d'un arbre réduit à une feuille.

Afin que les opérations d'insertion se fasse en temps logarithmique, il ne faut pas avoir à recalculer toutes les hauteurs à chaque étape (temps linéaire), mais simplement en temps proportionnel à la hauteur de l'arbre.

Exercice 50 Créez une nouvelle fonction `void accrocheAVL(noeud *pere, noeud *fils)` qui suit

la même logique que la fonction **accroche**. Il faut de plus modifier la hauteur (champ `valeur`) du père. Celle du fils ne change pas ! Pour cela, on regarde la hauteur de l'autre fils (que l'on suppose correcte). Si l'autre fils est vide, la hauteur du père vaut donc 1, sinon elle était déjà correctement initialisée par les ajouts précédents.

Exercice 51 Modifiez la fonction **getValeur** pour qu'elle retourne -1 quand l'arbre passé en paramètre est vide.

Exercice 52 Créez une fonction `void MAJHauteur(arbre *r)` qui calcule la hauteur du sommet r en fonction de la hauteur de ses fils, que l'on suppose bien initialisée. Cette fonction n'est pas récursive et utilise simplement la formule classique du calcul de la hauteur :

$$hauteur(r) = 1 + \max(hauteur(r.Fgauche), hauteur(r.Fdroit)).$$

On utilisera le champ `valeur` pour stocker la hauteur de l'arbre.

Exercice 53 Créez une nouvelle fonction `void insererDansAVL(arbre* r, noeud * n)`. Cette fonction va faire l'insertion des sommets comme dans l'arbre en mettant simplement à jour les hauteurs. On laissera une autre fonction le soin de rééquilibrer.

Exercice 54 Créez une fonction `void insererTableauAVL(arbre **pr, int * tab, int taille )` qui opère de la même façon que **insererTableau**. Cependant, le premier paramètre est un pointeur sur un arbre, ce qui nous permettra par la suite de modifier le sommet de l'arbre plus facilement. Pour l'instant, cette fonction ne fait qu'insérer les sommets un par un.

```
1   arbre *r4;
2   r4 = creerNoeudAVL(100);
3   insererTableauAVL(&r4, t, 16);
4   printf("En ne faisant pas de reequilbrage: \n");
5   afficherFonction(r4, 0, getCle);
6   printf("\n");
7   afficherFonction(r4, 0, getValeur);
8   printf("\n");
9   afficherFonction(r4, 0, difference);
10  printf("\n");
```

On obtient le résultat précédent, mais sans avoir à calculer les hauteurs (léger mieux !).

```
En ne faisant pas de reequilbrage:
100~~~110~~~120~~~130~~~140~~~150~~~160
|
0~~~~~10~~~~20~~~~30~~~~40~~~~50~~~~60~~~~70~~~~80~~~~90
10~~~~5~~~~~4~~~~~3~~~~~2~~~~~1~~~~~0
|
9~~~~~8~~~~~7~~~~~6~~~~~5~~~~~4~~~~~3~~~~~2~~~~~1~~~~~0
4~~~~~-5~~~~-4~~~~-3~~~~-2~~~~-1~~~~0
|
-9~~~~-8~~~~-7~~~~-6~~~~-5~~~~-4~~~~-3~~~~-2~~~~-1~~~~0
```

On va rajouter ce qui est nécessaire pour équilibrer après chaque insertion. Si vous avez suivi votre cours, les sommets à rééquilibrer sont sur le chemin entre la racine et le sommet nouvellement ajouté. Par ailleurs, il faut commencer par le sommet le plus en profondeur.

Exercice 55 Faites une fonction`void reequilibrerAVL(arbre ** pr, int cle)` qui permet de rééquilibrer l'arbre pour le remettre dans le droit chemin des AVL. Attention, la complexité de l'appel de cette fonction doit être linéaire en fonction de la distance entre la racine et le sommet étiqueté cle dans l'arbre (qui est une feuille).

Exercice 56 Complétez la fonction **`insererTableauAVL`** en faisant appel à la fonction précédente à chaque insertion.

Le résultat du programme précédent ressemble plus à ce que l'on attend ! On remarquera que l'AVL diffère légèrement de celui que l'on avait obtenu en rééquilibrant brutalement l'ABR initial.

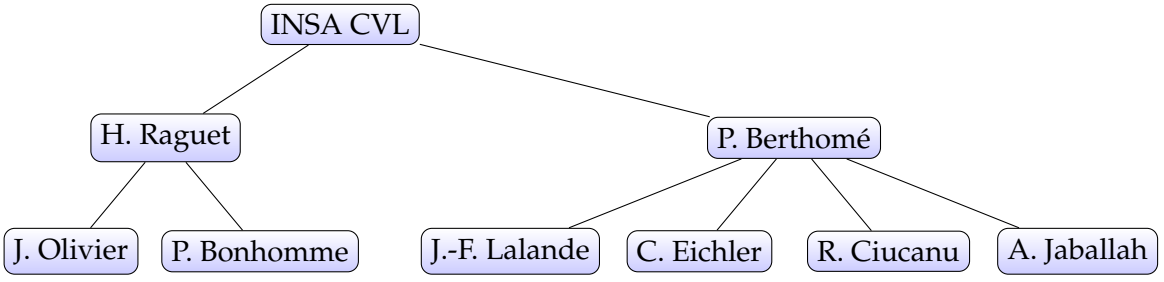
AVL
```
110~~~130~~~150~~~160
            |
            140
      |
      120
|
50~~~~90~~~~100
      |
      70~~~~80
      |
      60
|
30~~~~40
|
10~~~~20
|
0
```

Hauteur
```
4~~~~~3~~~~~1~~~~~0
            |
            0
      |
      0
|
3~~~~~2~~~~~0
      |
      1~~~~~0
      |
      0
|
2~~~~~0
|
1~~~~~0
|
0
```

Desequilibres
```
0~~~~~-1~~~~0~~~~~0
            |
            0
      |
      0
|
0~~~~~1~~~~~0
      |
      0~~~~~0
      |
      0
|
1~~~~~0
|
0~~~~~0
|
0
```

Challenge 15 Dé-commentez la fonction challenge_15 obtenue en suivant les instructions de l'annexe ⋆, ajoutez un appel à challenge_15 au début de votre fonction main, compilez et exécutez le programme. Collez l'affichage obtenu dans votre compte rendu, puis supprimer l'appel à challenge_15 dans le main.

Après cette étape, vous pouvez vous reposer avec le sentiment du devoir accompli !



Annexes

Liste des choses à ne pas oublier de faire :

⊘ Fonctions : prototypage et entêtes

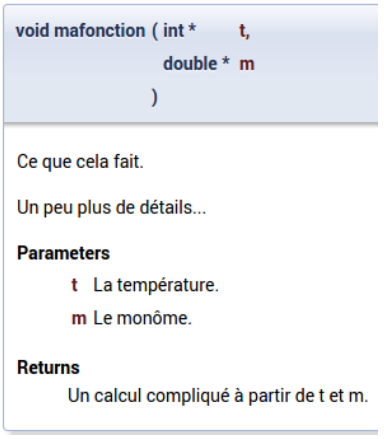
1. Prototyper vos fonctions dans vos .h

```
void mafonction(int * t, double * m);
```

```
void mafonction(int * t, double * m) { ... }
```

2. Documentez vos fonctions à l'aide des cartouches reconnues par doxygen :

```
/**
 * @brief Ce que cela fait.
 *
 * Un peu plus de details...
 *
 * @param t La temperature.
 * @param m Le monome.
 * @return Un calcul complique a partir
     de t et m.
 */
void mafonction(int * t, double * m)
{ ...
}
```



⋆ Les challenges : génération et exécution

Générez tous les challenges

Le site web accessible à l'adresse http ://172.30.128.8/ vous permettra de générer des challenges personnalisés à l'aide de l'interface suivante :

Ces challenges seront fonction de votre login INSA (usuellement composé de la première lettre de votre prénom suivi de votre nom). Commencez donc par renseigner votre login dans le champ *Username*.

Cliquez sur *Downlad all*. Vous récupérerez une archive contenant un .h et un .c contenant une fonction *evaluate* et 15 fonctions *challenge_i* pour i allant de 1 à 15.

Inclure tous les challenges générés dans votre projet

Ajoutez ces deux fichiers à votre projet. On notera des erreurs ! Certains challenges font appels à des fonctions que vous n'avez pas encore implémentées. On commentera tous les challenges pour commencer, puis les décommentera au fur et à mesure.

Exécuter le challenge i

Pour exécuter le challenge *i*, dé-commentez le dans le .h et .c générés via le site web et ajouté à votre projet dans l'étape précédente, puis utilisez simplement le code suivant dans votre fonction *main* :

```
1  int i = 1;
2  printf("\n-----------------------------------------\n");
3  printf("\tChallenge %i pour %s\n", i, "pberthom");
4  printf("\n-----------------------------------------\n");
5  challenge_i();
6  printf("-----------------------------------------\n");
7  printf("\tFin du Challenge %i\n", i);
8  printf("\n-----------------------------------------\n");
9  return 0;
```

On n'oubliera pas d'inclure l'affichage résultant dans le rapport !

```
-----------------------------------------
        Challenge 1 pour pberthom
-----------------------------------------
=> Calling accroche
J'accroche a gauche de 83 le noeud 80
J'accroche a gauche de 80 le noeud 2
J'accroche a droite de 2 le noeud 74
J'accroche a droite de 83 le noeud 178
J'accroche a gauche de 74 le noeud 52
J'accroche a gauche de 52 le noeud 40
J'accroche a gauche de 178 le noeud 120
J'accroche a droite de 74 le noeud 77
J'accroche a gauche de 120 le noeud 85
J'accroche a gauche de 40 le noeud 12
J'accroche a droite de 85 le noeud 89
J'accroche a droite de 178 le noeud 198
Le noeud 77 est deja present
J'accroche a droite de 52 le noeud 70
J'accroche a droite de 120 le noeud 155
J'accroche a droite de 89 le noeud 117
-----------------------------------------
        Fin du Challenge 1
-----------------------------------------
```