

T.P.3

Arbres

L'objectif de cette séance de TP est de réaliser en C les algorithmes présentés en cours.

1 Consignes

- Avant de passer à l'écriture d'une fonction numéro i , assurez vous que la fonction numéro $i - 1$ a été implémentée et testée (par un appel dans la fonction main).
- Réfléchissez bien aux prototypes des fonctions avant de les définir.

2 Les Arbres

L'ensemble de vos réalisations prendront place dans le fichier **tree.c** disponible dans l'archive ci-jointe. Pour vous aider dans la réalisation et la mise au point de vos algorithmes, nous vous fournissons les différents fichiers suivants :

1. **main.c** qui effectue des tests à partir des différentes fonctions que vous devez implémenter.
2. **tree.h** qui contient les prototypes des fonctions à réaliser et la définition de la structure d'arbre.
3. **iles.h** et **iles.c** qui offrent une implémentation dynamique des *pires* et *files* basée sur les listes : vous pouvez considérer ces deux fichiers comme un corrigé du TP2 (Files et Piles).
4. **list.h** et **list.c** qui offrent une implémentation des *listes* utilisée par nos piles et files.
5. **tree.cb** un fichier de projet CodeBlocks qui package l'ensemble des ressources et facilite la compilation.
6. **tree.txt** qui contient la trace d'exécution que vous devriez obtenir une fois votre TP terminé.

En outre, **tree.c** propose deux implémentations complètes qui vous aideront à démarrer :

- celle d'un algorithme qui construit un arbre de Syracuse de la profondeur souhaitée.
- deux fonctions d'affichage de la structure et du contenu de l'arbre.

Nous avons préféré manipuler des arbres d'entiers positifs ($T = \text{unsigned}$) plutôt que comportant des données de toute nature ($T = \text{void} *$). L'idée est de favoriser la compréhension de la logique de traitement plutôt que la généricité du code (plus générique, il serait aussi plus abscons).

Exercice 1 Vous devez compléter progressivement **tree.c** pour obtenir les implémentations correctes des algorithmes suivants :

1. Vérifier qu'un arbre représenté par l'adresse de sa racine, est vide.
2. Créer et initialiser un nœud d'arbre (arbre singleton).

3. Parcourir un arbre et afficher les éléments qu'il contient suivant un parcours :
 - A. itératif de gauche à droite, en largeur.
 - B. itératif de gauche à droite, en profondeur.
 - C. infixe de gauche à droite.
4. Libérer la mémoire occupée par un arbre.
5. Retourner le nombre d'éléments d'un arbre.
6. Retourner la hauteur d'un arbre.
7. Vérifier qu'un arbre b est le sous-arbre d'un arbre a .
8. Retourner une liste simplement chaînée qui représente la branche $[a, b]$ d'un arbre a si celle-ci existe, et une liste vide sinon.
9. Créer un arbre parfait de profondeur n dont les valeurs des nœuds sont exactement les index des nœuds suivant un parcours en largeur (index 1 pour la racine).
10. Vérifier qu'un arbre binaire est un arbre binaire de recherche. On peut proposer deux versions, une naïve, une performante. Pour la version naïve, il faut les auxiliaires de recherche des nœuds $\min$ et $\max$ d'un arbre.

Question+++ : Retourner la différence de profondeur entre la plus et la moins profonde feuille.