TP n°20 -Arbres Binaires de Recherche (ABR) en C - Approfondissement

Toutes les fonctions de ce TP seront codées dans le fichier nommé `NOM_bst.c` crée au TP précédent, sauf pour l'exercice 2. Bien entendu, vous ferez une copie de ce fichier dans le répertoire du nouveau TP pour redémarrer de ce travail mais en conservant une copie en l'état de ce fichier dans votre dossier du TP précédent.

Exercice 1 (Tri par ABR).
Un ABR peut être utilisé pour réaliser un tri.

1. Proposez une méthode permettant de trier un tableau en utilisant un ABR comme structure de données intermédiaire. Écrire l'algorithme en pseudo-code. *Indication : on pourra aller relire les réponses aux questions de l'exercice 1 du TP précédent.*
2. Écrire une fonction `void bst_sort(int size, int *tab)` qui trie un tableau d'entiers dont la taille est passée en paramètre en utilisant cette idée.
3. Quelle est la complexité dans le meilleur et dans le pire cas de ce tri ? Que dire de la complexité spatiale ?

Définition 1 (Dictionnaire)

Un **dictionnaire** est une structure de données qui permet d'associer à des clés (par exemple des clés de type chaîne de caractères) des valeurs.
À une clé correspond une unique valeur.

Remarque. Ajouter un nouveau couple (clé, valeur) pour une clé déjà existante écrase son ancienne valeur.
Ajouter un couple (clé, valeur) déjà existant est sans effet.

On pourra utiliser la fonction `strdup` (*string duplicate*) de la bibliothèque standard `string.h` dont le prototype est :

```
char *strdup(const char*s
```

Cette fonction crée une copie profonde [1] de la chaîne de caractères passée en argument, c'est-à-dire qu'elle alloue une nouvelle zone mémoire dans le tas avec `malloc`, correspondant à la taille de la chaîne donnée en entrée et copie la chaîne de caractère donnée en entrée dans ce nouvel espace mémoire.

Attention (Libération de la mémoire allouée par `strdup`). C'est au programmeur de libérer manuellement avec `free` toutes les chaînes de caractères allouées par l'intermédiaire de `strdup`.

On pourra également utiliser la fonction `strcmp` (*string compare*) de la bibliothèque standard `string.h` dont le prototype est :

```
int strcmp(const char*s1, const char*s2)
```

Cette fonction renvoie la valeur :

- 0 si les deux chaînes de caractères sont identiques,
- un nombre strictement négatif si `s1` est strictement inférieure à `s2` pour l'ordre lexicographique
- un nombre strictement positif si `s1` est strictement supérieure à `s2` pour l'ordre lexicographique

Attention (Caractère sentinelle `'\0'`). Je vous rappelle qu'en C, les chaînes de caractères sont des tableaux de caractères `char *` qui doivent impérativement se terminer par le caractère sentinelle `'\0'`. L'absence de ce caractère peut engendrer de graves dysfonctionnements dans toutes les fonctions de manipulation de chaînes de caractères, et en particulier toutes celles de la librairie `string.h` : la présence de ce caractère sentinelle en fin de chaîne de caractères est donc une précondition pour toutes les fonctions de manipulations de chaînes de caractères.

1. copie avec un espace mémoire indépendant, par opposition à une copie paresseuse, qui ne fait que copier l'adresse pointant vers le début de la chaîne

Exercice 2 (Implémentation d'une structure de dictionnaire avec un ABR).
Nous l'avons vu, un ABR permet de structurer efficacement des données dans la perspective d'une recherche. Si l'ABR est bien équilibré, toutes les opérations de base (recherche, insertion, suppression) seront logarithmiques par rapport aux nombres de nœuds de l'ABR. Cette structure de données hiérarchique est donc bien adaptée pour implémenter concrètement une structure de données abstraite de dictionnaire.

1. Copier le fichier original `bst.c` du TP précédent dans votre répertoire courant sous le nom `NOM_bst-dict.c` (on pourra également récupérer les fichiers corrigés des précédents exercices sur Moodle).
2. Adapter la structure de données `bst` pour pouvoir implémenter une structure de données de type dictionnaire avec des **clés de type chaînes de caractères** sous la forme d'un ABR.
3. Proposer une implémentation concrète de l'interface abstraite suivante :

```
bst *bstdict_create(bst *l, key_type k, elt_type v, bst *r);
void bstdict_free(bst **addr_t);
int  bstdict_number_of_entries(bst *t);
void bstdict_print_inorder(bst *t);
bst *bstdict_insert(key_type k, elt_type v, bst *t);
bool bstdict_find(key_type k, bst *t, node **addr_node);
void bstdict_remove(key_type k, bst *t);
```

Exercice 3 (ABR : fonction de vérification. Plus difficile.).
On repart du fichier `NOM_bst.c`.
On veut écrire une fonction récursive `bst_check` qui répond `true` si un arbre binaire est un ABR et `false` sinon.
On cherchera un algorithme qui effectue au plus un parcours de l'arbre.
Quelles informations doivent être remontées par chaque appel récursif pour permettre à l'appel du dessus de statuer sur le caractère ABR (ou non) ? Réfléchir sur papier.
Pour valider cette fonction, on reprendra les réponses du premier exercice du TP précédent pour créer un test pour lequel la réponse attendue est négative (arbre binaire qui n'est pas un ABR).