TP n°19 -Arbres Binaires de Recherche (ABR) en C

Toutes les fonctions de ce TP seront codées dans un fichier nommé `NOM_bst.c`.

Nous allons implémenter aujourd'hui en C la structure de données abstraite d'arbre binaire de recherche (ABR), appelée *binary search tree* (BST) en anglais.

Définition 1 (Arbre Binaire de Recherche (ABR))

On considère un arbre binaire étiqueté dont les étiquettes appartiennent à un ensemble E, et on considère une **relation d'ordre totale** $\preceq$ sur E.

Cet arbre binaire vérifie la propriété **d'arbre binaire de recherche** si l'étiquette d'un nœud est supérieure à toutes les étiquettes de tous les nœuds de son sous-arbre gauche et inférieure à toutes les étiquettes de tous les nœuds de son sous-arbre droit, pour la relation d'ordre totale choisie.

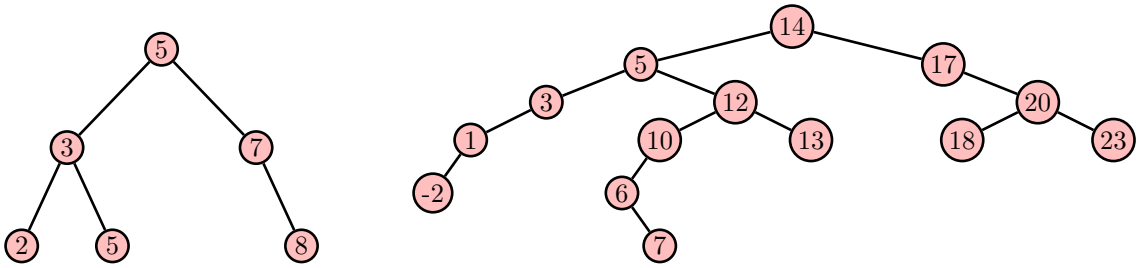
Autrement dit, soit x un nœud et $e(x) \in E$ son étiquette.
Pour tout nœud y du sous-arbre de gauche de x, $e(y) \preceq e(x)$.
Pour tout nœud y du sous-arbre droit de x, $e(x) \preceq e(y)$.

Remarques.

- On pourrait imposer une inégalité stricte à gauche et/ou à droite. En pratique, on utilise souvent les arbres binaires de recherche pour stocker des étiquettes (ou clés) sans doublons. **Nous considérerons que nos arbres binaires n'ont pas d'étiquettes en doublon dans toute la suite de ce TP.**
- Pour ce premier TP sur les ABR, nous allons considérer des ABR à étiquettes entières et on munit cet ensemble de la relation d'ordre usuelle $\leq$: $(E, \preceq) = (\mathbb{Z}, \leq)$

Exemple 1

Par exemple, les arbres suivants sont des arbres binaires de recherche.



Exercice 1 (Quelques remarques préliminaires).

1. Dessiner proprement l'ABR obtenu en insérant successivement les nœuds d'étiquettes suivantes : 15, 5, 16, 3, 12, 20, 10, 13, 18, 23, 6, 7
2. Dessiner proprement l'ABR obtenu en insérant successivement les nœuds d'étiquettes suivantes : 5, 15, 16, 3, 12, 20, 10, 13, 18, 23, 6, 7. Quelle observation faites-vous ? Écrire une phrase !
3. Quel parcours permet d'afficher toutes les étiquettes de l'arbre dans l'ordre croissant ?
4. Pour qu'un arbre binaire soit un ABR, il ne suffit pas de supposer que les sous-arbres gauche et droit le sont et que l'étiquette de la racine est supérieure à l'étiquette de la racine du sous-arbre gauche et inférieure à celle de la racine du sous-arbre droite. Dessiner un contre-exemple.
5. Comment reconnaît-on à coup sûr le nœud d'étiquette minimale d'un sous-arbre dans un ABR ?
6. Quelle est la complexité temporelle d'une opération de recherche d'une valeur dans un ABR dans le pire des cas ? Dans le meilleur des cas ?

Exercice 2 (ABR : implémentation des fonctions de manipulation basiques).

1. Téléchargez le fichier `bintree-ex1et2.c` corrigé du dernier TP correspondant aux exercices 1 et 2 sur Moodle et renommez-le `NOM_bst.c`
2. En utilisant la commande `emacs Alt+Maj + %`, remplacez toutes les occurrences de la chaîne de caractère `bintree` par la chaîne `bst` dans votre fichier source. Il faut appuyer sur la touche `Y` pour valider chaque remplacement de l'ancienne chaîne par la nouvelle chaîne.
3. Modifier la structure de données pour pouvoir stocker l'adresse du nœud parent de chaque nœud. On nommera `parent` ce nouveau champ. Modifier les fonctions existantes en conséquence, lorsque cela est nécessaire.
4. Écrire une fonction récursive `bst *bst_insert(elt_type k, bst *t)` permettant d'insérer une nouvelle valeur dans un ABR en respectant la propriété d'ABR.
5. Dans le `main`, créer l'ABR donné en exemple ci-dessus à droite.
6. Écrire une version itérative `bst_insert_iter` de cette fonction d'insertion
7. Écrire une fonction récursive `elt_type bst_min(bst *t)` qui renvoie l'étiquette minimale d'un ABR.
8. Améliorer la fonction précédente pour qu'elle renvoie un pointeur vers le nœud d'étiquette minimale `node *bst_min(bst *t)`
9. Écrire une version itérative `node *bst_min_iter(bst *t)` de la fonction précédente.
10. Écrire une fonction itérative qui retourne *faux* si une valeur donnée n'est pas présente dans l'ABR, et *vrai* sinon. Le prototype de la fonction est donc :

    ```
    bool bst_find_iter(elt_type k, bst *t)
    ```

11. Améliorez votre fonction itérative pour que, dans le cas où la valeur k est présente dans l'ABR, elle renvoie en plus l'adresse du nœud trouvé, en utilisant un passage par adresse. Le prototype de la fonction sera :

    ```
    bool bst_find_iter(elt_type k, bst *t, node **addr_node)
    ```

12. Écrire une version récursive `bst_find` de cette fonction, avec le même prototype.

Exercice 3 (ABR : suppression d'un nœud).

Supposons désormais que l'on cherche à supprimer un nœud `z` de l'ABR. Trois cas sont à envisager, cf Figure 1 :

(a) Si `z` est une feuille, c'est-à-dire si ses deux fils sont `NULL`, alors on peut directement supprimer `z` ;

(b) Si `z` n'a qu'un seul fils, alors on peut « détacher » `z` en reliant ce fils avec le parent de `z` ;

(c) En revanche, si `z` a deux fils la situation est plus complexe. Nous avons vu que l'on peut alors remplacer `z` par le minimum de son fils droit (ou par le maximum de son fils gauche) que l'on note `y`. On peut recopier la clé de `y` dans `z` puis supprimer `y`. Comme `y` est le minimum d'un sous-arbre, son fils gauche est nécessairement [a] vide : on se retrouve dans le deuxième cas que l'on sait gérer.

Écrire une fonction `bst_remove` qui supprime le nœud ayant une étiquette donnée k dans l'ABR. On suppose qu'il n'y a pas de doublon.
Indication : pour savoir si un nœud est le fils gauche ou bien le fils droit de son parent, on pourra procéder en comparant l'adresse du nœud avec l'adresse des fils gauche et droit de son parent.

a. Il peut y avoir plusieurs réalisations du minimum, mais il est toujours possible d'en choisir une qui n'a pas de fils gauche. C'est *a priori* bien le cas de notre fonction `bst_min`. Vérifiez-le.

Exercice 4 (A la maison : amélioration de la complexité spatiale).

Faire une sauvegarde du code réalisé `bst.c` dans un fichier `bst-with-parent.c`
Reprendre le code `bst.c` et retirez le stockage de l'adresse du nœud parent dans la structure de donnée. Adapter tout le code en conséquence.
Indication : certaines fonctions devront maintenant renvoyer, en plus du nœud trouvé, le nœud parent du nœud trouvé. Cela se fera en réalisant un passage par adresse comme celui qui a été fait dans la fonction `bst_find` pour renvoyer l'adresse du nœud trouvé.

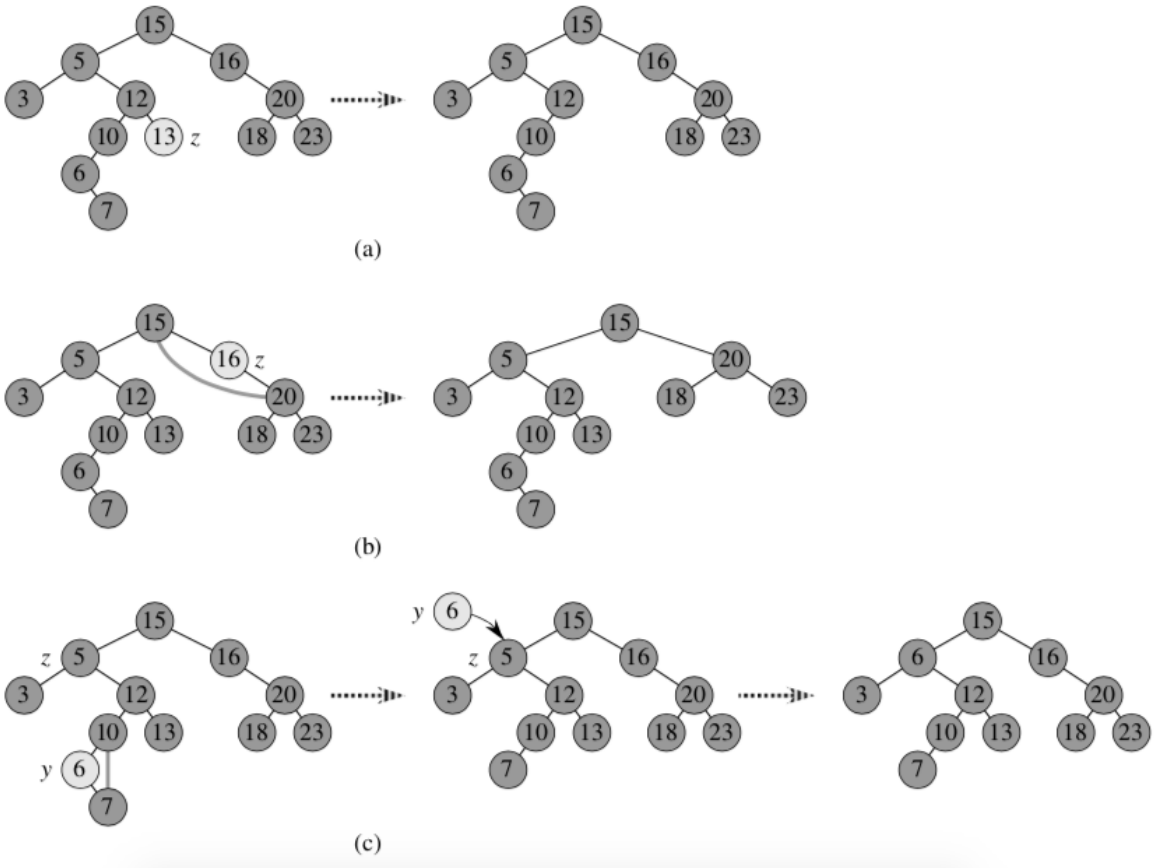


FIGURE 1 – Suppression d'un nœud `z` dans un arbre binaire de recherche. Le nœud à supprimer effectivement est colorié en gris clair. (a) `z` n'a pas d'enfants : on peut le supprimer directement. (b) `z` n'a qu'un seul enfant : on peut le détacher. (c) `z` a deux enfants : on le remplace par le minimum de son sous-arbre droit, appelé ici `y`. Ce nœud `y` n'a pas de fils gauche et on peut donc le détacher.