1 Petits exercices sur les ABR

On considère la déclaration suivante :

```
struct noeud {
    struct noeud* gauche;
    struct noeud* droit;
    int valeur;
};
#define NIL (struct noeud*)0
```

Question 1. Insérer les entiers suivants dans un ABR initialement vide, dans l'ordre. Détailler les modifications faites à l'arbre. Quel est le nombre de nœuds de l'ABR, sa hauteur ? quelles sont ses feuilles, sa racine ? L'ABR est-il équilibré en nombre de nœuds ? en hauteur ?

14, 7, 88, 51, 17, 53, 3 .

Question 2. Écrire une fonction qui retourne `true` si l'ABR qui lui est passé en paramètre est une feuille.

Question 3. Écrire une fonction récursive qui imprime l'ABR qui lui est passé en paramètre, par ordre croissant des valeurs.

Question 4. Écrire une fonction récursive qui imprime l'ABR qui lui est passé en paramètre, par ordre décroissant des valeurs.

Question 5. Écrire un destructeur d'ABR.

Question 6. Écrire une fonction qui imprime toutes les valeurs des nœuds, en les indentant en fonction de la profondeur du nœud, dans l'arbre (la valeur de la racine en première colonne, celles des fils de la racine en colonne 4, celles des petits-fils en colonne 8, etc.).

Question 7. Écrire une fonction récursive qui imprime un ABR dans un fichier `ABR.dot`, au format du logiciel `dot`, qui construit un fichier `ABR.pdf` grâce à la commande `system`, puis qui visualise l'arbre. Vous pouvez vous aider d'une fonction auxiliaire. Par exemple, le fichier `ABR.dot` suivant :

```
digraph G {
    56 -> 15 [label="gauche"];
    15 -> 4 [label="gauche"];
    4 -> 5 [label="droit"];
    5 -> 8 [label="droit"];
    15 -> 46 [label="droit"];
    56 -> 81 [label="droit"];
};
```

permet d'afficher le graphique de la Figure 1, en enchaînant les commandes :

```
$ dot -Tpdf ABR.dot -Grankdir=LR -o ABR.pdf
$ evince ABR.pdf
```

Dans le cas d'un arbre réduit à une feuille, le fichier `ABR.dot` ressemblerait à :

```
digraph G {
    56;
};
```

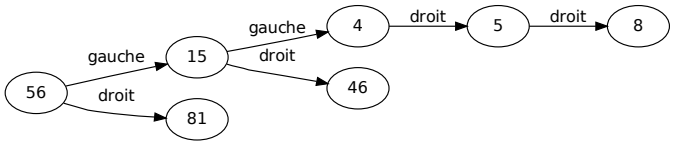

FIGURE 1 – Un ABR visualisé avec `dot`.

Question 8. Écrire une fonction qui retourne la hauteur de l'ABR qui lui est passé en paramètre.

Question 9. Écrire une fonction qui retourne le nombre de nœuds de l'ABR qui lui est passé en paramètre.

Question 10. Écrire une fonction `ieme`, paramétrée par un ABR A, un entier i et qui retourne le ième élément de A (les éléments sont numérotés à partir de 0).

Question 11. Proposer une modification de l'implantation des ABR qui permette d'améliorer le temps de calcul de la fonction `ieme`.

2 Parcours d'arbres (et de graphes) en espace constant

Certains langages de programmation offrent la possibilité d'allouer des blocs mémoire (par une fonction analogue à *malloc*) sans jamais devoir la libérer. Il peut alors être utile de disposer d'un algorithme de parcours d'arbre (ou même plus généralement de graphe) qui ne consomme pas de mémoire (de tels algorithmes sont déclenchés lorsque la mémoire est saturée) :

— l'algorithme ne doit pas être récursif (les appels récursifs consomment de la mémoire dans la pile d'exécution du processus) ;

— l'algorithme ne doit pas utiliser de structures de données susceptibles de consommer de la mémoire dynamiquement (piles, files, listes ...).

On décrit ici, pour le cas des arbres binaires, une solution qui a été développée pour les environnements d'exécution de langages tels que LISP. L'idée consiste à « retourner » les pointeurs `gauche` ou `droit` des nœuds lorsqu'on descend dans les arbres et à les « restaurer » lorsqu'on remonte.

Lors du parcours, on utilise deux variables locales (des pointeurs de nœuds) : `cour` pointe sur le nœud courant et `prec` pointe sur le prédecesseur de `cour`. Les figures 2, 3 et 4 illustrent le mécanisme de parcours.

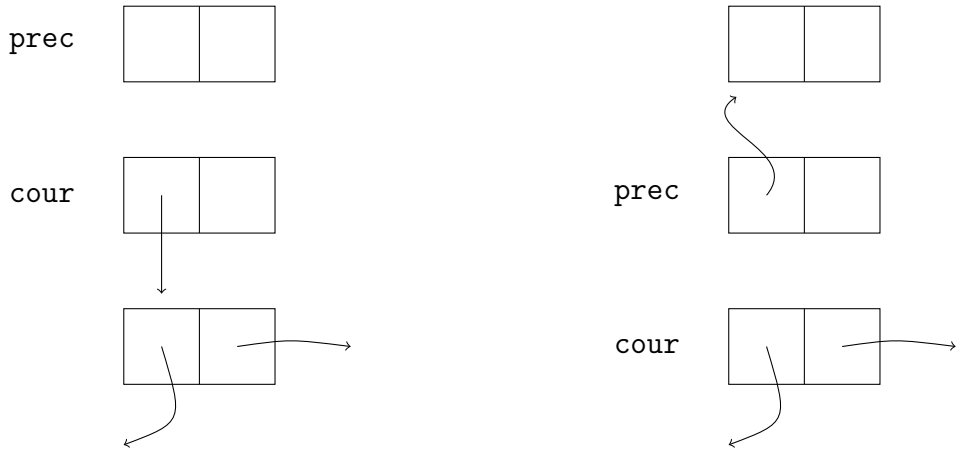

FIGURE 2 – Cas d'une descente suivant `cour->gauche`. À gauche : avant la descente ; à droite : après la descente. Les pointeurs `prec` et `cour` descendent tous les deux. Le pointeur gauche du nœud central est « retourné » pour permettre de remonter à une étape ultérieure. On effectue une telle descente si `cour` est différent de `NIL` et n'a pas encore été exploré.

Un pseudo-code est donné Figure 5. Pour l'implantation, on peut ajouter à chaque nœud deux booléens :

— `noeud_explore`, initialement faux, indique si le nœud a déjà été exploré ou est en cours d'exploration (on pourrait se passer de ce booléen pour les parcours d'arbres mais il est utile pour les parcours de graphes) ;

— `sous_arbre_droit_explore`, initialement faux, indique si le sous-arbre droit du nœud a déjà été exploré ou est en cours d'exploration.

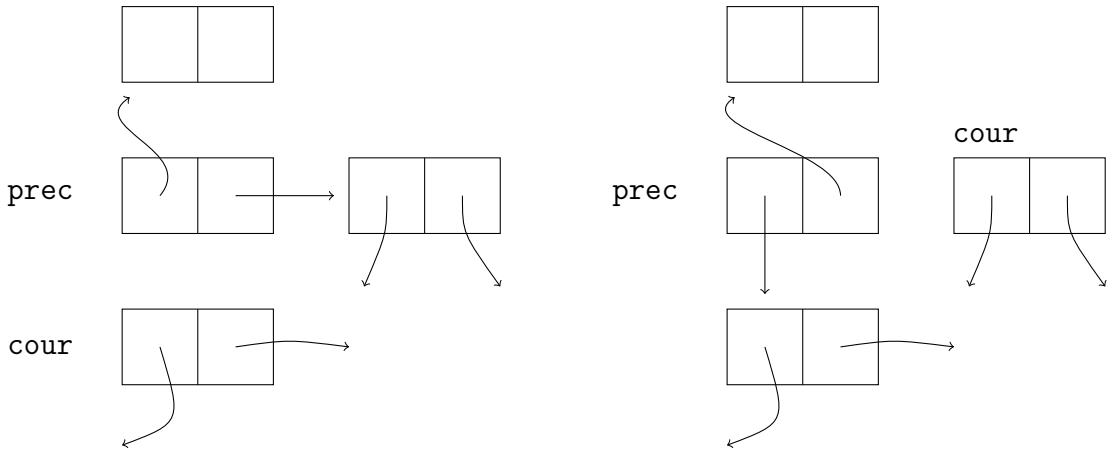

FIGURE 3 – Cas d'une descente suivant `prec->droit`. À gauche : avant la descente ; à droite : après la descente. On effectue cette descente lorsqu'on ne veut plus descendre à gauche (cas où `cour` vaut `NIL`, a déjà été ou a fini d'être exploré) et que le sous-arbre droit de `prec` n'a pas encore été exploré. Le pointeur `prec` ne bouge pas. Avant de déplacer `cour`, on « restaure » `prec->gauche`. Le pointeur `prec->droit` est « retourné » pour permettre de remonter à une étape ultérieure.

Question 12. Coder l'algorithme de la Figure 5 en C.

Question 13. Insérer une instruction d'affichage à la bonne place, pour effectuer un affichage par valeur croissante (dans le cas d'un ABR).

Question 14. Comment faudrait-il modifier le code pour effectuer un affichage par valeur décroissante (dans le cas d'un ABR) ?

Question 15. Dans le cas d'un parcours d'arbre, il est possible de remettre à faux les booléens lors des remontées, ce qui permet d'enchaîner plusieurs parcours d'affilée. Effectuer la modification.

Question 16. Pour gagner de la place en mémoire, les deux booléens peuvent être codés en marquant à 1 un bit « inutilisé » dans les pointeurs gauche et droit de chaque nœud (par exemple le bit de poids faible de chaque pointeur vaut nécessairement zéro en raison de contraintes d'alignement de structures en mémoire : il peut donc être utilisé pour coder un booléen). Lors des modifications de pointeurs, attention au fait que les booléens sont censés être attachés aux nœuds, pas aux pointeurs.

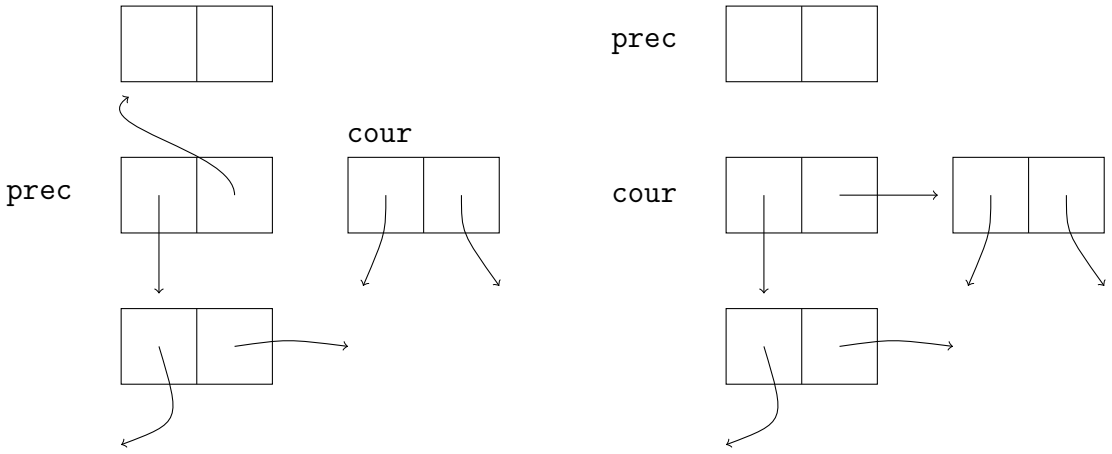

FIGURE 4 – Cas d'une remontée. À gauche : avant la remontée ; à droite : après la remontée. On effectue cette remontée lorsqu'on ne veut plus descendre ni à gauche, ni à droite. Comme on explore les sous-arbres gauches avant les sous-arbres droits, cette situation se produit à la fin de l'exploration du sous-arbre droit de `prec` : on doit donc remonter en suivant `prec->droit`. Avant de remonter `cour`, on « restaure » `prec->droit`.

```
procedure parcours (racine)
begin
  prec := NIL
  cour := racine
  while prec ≠ NIL ou cour ≠ NIL et n'est pas marqué comme exploré do
    if cour n'est pas marqué comme exploré then (Figure 2)
      marquer cour comme exploré
      tmp := cour→gauche
      cour→gauche := prec
      prec := cour
      cour := tmp
    elif prec ≠ NIL et n'est pas marqué
                          comme ayant son sous-arbre droit exploré then (Figure 3)
      marquer prec comme ayant son sous-arbre droit exploré
      tmp := prec→droit
      prec→droit := prec→gauche
      prec→gauche := cour
      cour := tmp
    elif prec ≠ NIL then (Figure 4)
      tmp := prec→droit
      prec→droit := cour
      cour := prec
      prec := tmp
    fi
  end do
end
```

FIGURE 5 – Algorithme de parcours. Les tests doivent être traduits avec soin.