Module ITC313 - Informatique

Partie C / C++

TP3 + TP4 manipulation des arbres

Benoît Darties - benoit.darties@u-bourgogne.fr
Université de Bourgogne

Année universitaire 2016-2017

1 Algorithmes sur arborescences binaires de recherche (ABR) non équilibrées

Dans ce TP nous mettons en place une nouvelle structure, les Arbres Binaires de Recherche (ABR). Les ABR, tout comme les listes chaînées, permettent un stockage optimal en mémoire lorsque le nombre d'éléments qu'ils contiennent varie. Mais ils offrent en plus de meilleures performances pour la recherche de valeurs contenues, proposant des algorithmes en ($O(\log(n))$ là où les listes chaînées nécessitent des algorithmes de comparaisons en $0(n)$. Pour rappel, un ABR contient des nœuds, organisés en arborescence binaire : chaque nœud contient une valeur entière, ainsi que deux pointeurs sur chacun de ses 2 fils, respectivement nommés `filsGauche` *et* `filsDroit`*. Les valeurs sont ordonnées de sorte que toutes les valeurs situées dans le sous-arbre gauche d'un noeud sont inférieures à la valeur d'un noeud, et toutes les valeurs situées dans le sous-arbre droit sont supérieures à la valeur du noeud. Le premier nœud de l'arbre est appelé* `racine`*. Pour accéder à tous les éléments de l'arbre, il suffit donc d'avoir constamment un pointeur sur la racine de l'arbre. Ce TP se présente comme un fil rouge, les codes développés dans un exercice étant par la suite réutilisés dans le TP suivant. Chaque fonction développée dans un exercice est donc réutilisable dans l'exercice suivant.*

Exercice 1 : *Mise en place d'ABR et premiers algorithmes*

Dans cet exercice, nous définissions les éléments primordiaux à la mise en place d'une ABR, ainsi que certains algorithmes classiques.

On vous fournit dans cet exercice les fichiers `main.c`, `arbresbasiques.h` et `arbresbasiques.c`. L'objectif est de mettre en place les éléments de base permettant de définir des arbres. Pour vous aider, on vous fournit déjà deux fonctions dans le fichier `arbresbasiques.c` :

- La fonction `unArbrePredefini()` : cette fonction renvoie un pointeur vers la racine d'un arbre représenté en mémoire par la figure 1. Cette fonction sera effective une fois les questions 1 et 2 de l'exercice correctement effectuées.
- La procédure `afficheArbre()` : cette procédure affiche un arbre en listant chacun de ses noeuds et, et chacun des fils de ces noeuds. Pour l'arbre présenté dans la Figure 1 , la fonction affichera le résultat présenté dans le listing ci-après.
  Cette fonction est d'ores et déjà opérationnelle.

Nous n'avez pas besoin de modifier ces deux fonctions, mais les utiliserez pour tester rapidement les résultats des fonctions que vous avez créés. On vous fournit également dans le fichier `main.c` le code de la fonction `main()` qui vous permet de tester les fonctions une fois ces dernières définies.

1. Ouvrez le fichier `arbresbasiques.h` et éditez le squelette de la structure `noeud`. Ajoutez y les éléments suivants *en respectant les noms des variables* : l'élément `valeur`, de type `int`, qui

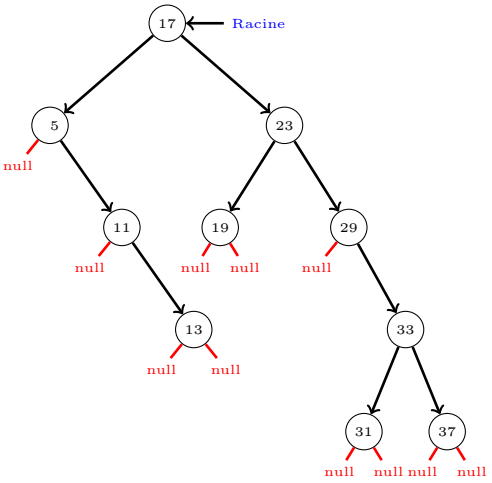

FIGURE 1 – Arbre renvoyé par la fonction `unArbrePredefini()`

Listing 1 – affichage de l'arbre avec la fonction afficheArbre()

```
1  Noeud   17 : filsGauche :    5  - filsDroit :   23
2  Noeud    5 : filsGauche : vide  - filsDroit :   11
3  Noeud   11 : filsGauche : vide  - filsDroit :   13
4  Noeud   13 : filsGauche : vide  - filsDroit : vide
5  Noeud   23 : filsGauche :   19  - filsDroit :   29
6  Noeud   19 : filsGauche : vide  - filsDroit : vide
7  Noeud   29 : filsGauche : vide  - filsDroit :   33
8  Noeud   33 : filsGauche :   31  - filsDroit :   37
9  Noeud   31 : filsGauche : vide  - filsDroit : vide
10 Noeud   37 : filsGauche : vide  - filsDroit : vide
```


Remarques / questions : benoit.darties@u-bourgogne.fr

contiendra la valeur à stocker ; ainsi que deux éléments `filsGauche` et `filsDroit`, de type *pointeur sur une structure de type* ***`noeud`***, qui pointeront respectivement vers le fils gauche et le fils droit du nœud.

2. Ouvrez le fichier `arbresbasiques.c` et ajoutez-y une fonction `nouveauNoeud()`. Cette fonction prend en paramètre une valeur, et réserve la mémoire nécessaire au stockage d'un nouveau nœud dans le tas. Elle affecte alors l'élément `valeur` avec la valeur passée en paramètre, et fait pointer `filsGauche` et `filsDroit` vers `null`. Enfin, elle retourne un pointeur vers
3. Compilez alors votre programme avec la commande :

   ```
   gcc arbresbasiques.c main.c -o ./arbresDebut
   ```

   et lancez le programme `arbresDebut` pour vérifier que la définition de la structure et de la fonctions sont correctes. Nous devriez normalement visualiser quelque chose de similaire au précédent listing.
4. Ajoutez dans `arbresbasiques.c` une fonction `estvide()`. Cette fonction prend en paramètres un pointeur sur un noeud de l'arbre, et renvoie 1 si le pointeur a comme valeur NULL, et 0 sinon. Cette fonction permet par la suite de déterminer plus facilement si un sous-arbre est vide.
5. Ajoutez une fonction `estFeuille()`. Cette fonction prend en paramètres un pointeur sur un noeud de l'arbre, et renvoie 1 si le noeud pointé est une feuille (le noeud existe et ses deux fils pointent sur NULL), et 0 sinon.
6. Ajoutez une fonction itérative `rechercheValeur()`. Cette fonction prend deux paramètres, le premier étant un pointeur sur la racine de l'arbre, et le second une valeur `v` à chercher. La fonction renvoie 1 si la valeur `v` est contenue dans l'arbre, et 0 sinon. Le déroulement de la fonction est le suivant : un pointeur `parcours` pointe sur le premier élément de l'arbre. Si cet élément est vide (pointeur null), la valeur `v` n'existe pas, et on renvoie 0. si cet élément existe, on regarde sa valeur. Si cette valeur est la valeur recherchée, on renvoie 1. Sinon, on déplace le pointeur `parcours` vers le fils gauche ou le fils droit du noeud selon que la valeur recherchée strictement inférieure ou strictement supérieure à la valeur du noeud pointé. Et on recommence.

Exercice 2 : *Algorithmes récursifs sur arborescences*

Dans cet exercice, nous ajoutons quelques algorithmes récursifs sur les arbres.

Pour cet exercice, vous reprendrez les fichiers `arbresbasiques.h` et `arbresbasiques.c` que vous avez complété dans l'exercice précédent. On vous fournit ici les fichiers `arbresFctRecursives.c` et `arbresFctRecursives.h` contenant les squelettes des fonctions récursives à définir ici, ainsi que le fichier `mainExo2.c` contenant la fonction `main()` et les appels pour tester les fonctions définies. Seuls le fichier `arbresFctRecursives.c` sera à éditer dans cet exercice, les tests pour les fonctions étant déjà écrits dans la fonction `main()`.

1. Pour compiler votre programme sous le nom `arbresRecursifs` en incluant les fonctions définies dans le fichier `arbresFctRecursives.c`, vous utiliserez la ligne de commande :

   ```
   gcc arbresbasiques.c arbresFctRecursives.c mainExo2.c -o ./arbresRecursifs
   ```

   Compilez une première fois le programme pour vous assurer que tout fonctionne correctement.
2. Ajoutez dans `arbresFctRecursives.c` une fonction récursive `nbNoeuds()`. Cette fonction prend en paramètre un pointeur `noeudCourant` vers le premier nœ ud d'un arbre, et renvoie son nombre de noeuds. Pour écrire cette fonction de manière récursive, il suffit de noter que :
   — Le nombre de noeuds d'un arbre vide (c.a.d. `noeudCourant` est égal à `NULL`) est égal à 0.
   — Sinon, il est égal à 1 + le nombre de noeuds de son sous-arbre droit + le nombre de noeuds de son sous-arbre gauche.
3. Ajoutez dans `arbresFctRecursives.c` une fonction récursive `sommeValArbres()`. Cette fonction prend en paramètre un pointeur `noeudCourant` vers le premier nœ ud d'un arbre, et renvoie la somme des valeurs des noeuds. Pour écrire cette fonction de manière récursive, il suffit de noter que :

— La somme des noeuds d'un arbre vide (c.a.d. `noeudCourant` est égal à `NULL`) est égal à 0.
— Sinon, elle est égale à la valeur contenue dans le noeud + la somme des noeuds de son sous-arbre droit + la somme des noeuds de son sous-arbre gauche.

4. Ajoutez ensuite une fonction récursive `hauteur()`. Cette fonction prend en paramètre un pointeur `noeudCourant` vers le premier nœ ud d'un arbre, et renvoie sa hauteur. Pour écrire cette fonction de manière récursive, il suffit de noter que :
   — La hauteur d'un arbre vide (c.a.d. `noeudCourant` est égal à `NULL`) est égal à 0.
   — Sinon, elle est égale au maximum des profondeurs de chacun de ses sous-arbres + 1
5. Ajoutez enfin une procédure récursive `detruireArbre()`. Cette fonction prend en paramètre un pointeur vers la racine de l'arbre `racine`, et détruit l'arbre en libérant sa mémoire. Avant de libérer la mémoire d'un noeud, il faut bien évidemment libérer la mémoire de chacun de ses sous-arbres.

Testez les fonctions ajoutées en utilisant la fonction `mainExo2()` qui vous est donnée. Puis modifiez le programme pour tester ces fonctions sur des arborescences que vous aurez généré aléatoirement.

2 Modification et parcours d'ABR non équilibrées

Exercice 3 : *Insertion et suppression de valeurs dans une arborescence*

Dans cet exercice, nous ajoutons quelques algorithmes permettant d'ajouter et supprimer des noeuds sur les arbres.

Pour cet exercice, vous reprendrez les fichiers `arbresbasiques.h` et `arbresbasiques.c` que vous avez complété dans l'exercice 1, ainsi que les fichiers `arbresFctRecursives.c` et `arbresFctRecursives.h` que vous avez complété dans l'exercice 2. On vous fournit ici les fichiers `modifArbres.c` et `modifArbres.h` contenant les squelettes des fonctions récursives à définir ici, ainsi que le fichier `mainExo3.c` contenant la fonction `main()` et les appels pour tester les fonctions définies.

1. Pour compiler votre programme sous le nom `arbresModifs` en incluant les fonctions définies dans le fichier `arbresFctRecursives.c`, vous utiliserez la ligne de commande :

   ```
   gcc arbresbasiques.c arbresFctRecursives.c modifArbres.c mainExo3.c -o ./arbresModifs
   ```

2. Ajoutez dans `modifArbres.c` le contenu de la fonction `ajouterValeurABR()`. Cette fonction prend comme paramètres un pointeur `racine` vers le premier nœ ud d'un arbre ainsi qu'une valeur, et ajoute cette valeur à l'arbre. De plus, elle renvoie un pointeur vers le premier noeud de l'arbre, au cas où ce dernier aurait changé (ce qui est le cas si l'arbre était vide). Rappelons que l'on ajoute un noeud en tant que nouvelle feuille, à l'unique endroit possible pour maintenir le caractère *de recherche* de l'arbre binaire. Pour vous aider, le détail du fonctionnement de cette fonction est présenté dans les commentaires du fichier `modifArbres.c`. Il vous suffit de rajouter les instructions correspondant aux différents commentaires.
3. Ajoutez dans `modifArbres.c` le contenu de la fonction `supprimerValeurABR()`. Cette fonction prend comme paramètres un pointeur `racine` vers le premier nœ ud d'un arbre ainsi qu'une valeur, et supprime cette valeur de l'arbre si cette dernière existait. De plus, elle renvoie un pointeur vers le premier noeud de l'arbre, au cas où ce dernier aurait changé (ce qui est le cas si le premier noeud était la valeur à supprimer). Rappelons que la suppression d'une valeur d'un noeud doit maintenir le caractère *de recherche* de l'arbre binaire. Pour vous aider, le détail du fonctionnement de cette fonction est présenté dans les commentaires du fichier `modifArbres.c`. Il vous suffit de rajouter les instructions correspondant aux différents commentaires.

Remarques / questions : benoit.darties@u-bourgogne.fr

3 Parcours d'arborescences binaires

Exercice 4 : *Parcours sur arbres*

Le parcours de valeurs sur un arbre est différent de ce que l'on peut faire sur une chaîne ou un tableau. Nous allons voir dans cet exercice les prémices d'un concept qui sera plus amplement développé en C++ : les itérateurs.

Si l'on souhaite lister tous les éléments d'une liste chainées d'un tableau, ou d'un arbre, il est possible de définir des fonctions itératives ou récursives (comme c'est le cas pour les arbres) permettant de répondre au problème. Mais si le parcours d'éléments sur les listes et tableaux est assez immédiat, celui des arbres peut se montrer plus délicat. Par exemple : dans quel ordre faut-il afficher les éléments. Il existe différents parcours permettant de lister les éléments d'un arbre. Les deux plus connus sont :

— le parcours en largeur : les éléments apparaissent successivement en fonction de leur profondeur
— le parcours en profondeur : avant d'afficher chaque élément, on doit avoir affiché son sous-arbre gauche, et on affichera son sous-abre droit après

Nous ne mettrons en place dans cet exercice que le parcours en profondeur (plus facile et plus utilisé).

Comme pour les autres exercices, vous reprendrez dans cet exercice les différents fichiers des précédents exercices, et y ajouterez les fichiers `parcoursArbres.c` et `parcoursArbres.h` contenant les squelettes des fonctions à définir ici, ainsi que le fichier `mainExo4.c` contenant la fonction `main()` et les appels pour tester les fonctions définies.

1. Définissez la bonne commande pour compiler efficacement le programme sous le nom `parcours` en prenant en compte les nouveaux fichiers
2. complétez la fonction récursive `parcoursProfondeur()` permettant de réaliser un parcours en profondeur, c'est à dire lister les éléments dans l'ordre suivant : *(sous-arbre gauche) element (sous-arbre droit)*. Vous pourrez vous inspirer de la fonction `afficheArbre()` si vous le souhaitez, et des commentaires présents dans le squelette de la fonction. Que constatez-vous en affichant ce parcours sur une arborescence binaire ?

Une des principales difficultés des arborescences par rapport aux chaines et tableaux est de trouver l'élément suivant d'un parcours, à partir d'un emplacement donné. Par exemple, dans une chaine, l'élément suivant d'un parcours depuis un élément donné est l'élément pointé par `next`. Dans un tableau, il s'agit de l'élément d'indice suivant. Dans un arbre binaire, ceci est plus délicat : vous pouvez vous en apercevoir en reprenant le parcours en profondeur généré sur l'exemple de la figure 1 : voyez que l'élément qui doit suivre 11 est son fils droit, à savoir 13. En revanche, l'élément suivant est 17, qui n'a aucun lien avec 13, et l'élément qui suit 17 est 19, qui n'a aucun lien avec 17. Ce que nous allons mettre en place ici est un itérateur, c'est à dire une méthode qui, à partir d'un élément courant, va retourner l'élément suivant du parcours.

Les conditions à appliquer pour définir cet itérateur sont les suivantes :

— le premier élément retourné par l'itérateur est l'élément le plus à gauche (tant qu'il existe un fils gauche, on avance. On s'arrête sur celui qui n'a plus de fils gauche, meme s'il a un fils droit qui lui meme a un fils gauche)
— si l'élément courant possède un fils droit, alors on prend la direction de ce fils droit, puis on avance tout à gauche.
— si l'élément courant ne possède pas de fils droit, alors on remonte jusqu'au premier noeud parent dans l'arbre pour lequel on venait de gauche
— le dernier élément retourné par l'itérateur est l'élément le plus à droite (tant qu'il existe un fils droit, on avance. On s'arrête sur celui qui n'a plus de fils droit)

3. Appliquez au stylo le principe de l'itérateur sur l'exemple de la figure 1, et vérifiez que vous obtenez le même parcours que celui obtenu avec la fonction `parcoursProfondeur()`.
4. Le principal frein a la mise en place de l'algorithme des itérateurs est l'impossiblité de "remonter" dans l'arbre, en accédant au parent d'un noeud à partir de ce dernier. Pour remédier

Remarques / questions : benoit.darties@u-bourgogne.fr

à cela, il va nous falloir modifier **tout notre code** pour intégrer cette nouvelle donnée.

(a) Modifiez la structure `noeud` du fichier pour qu'elle comprenne désormais un nouveau paramètre : le parent d'un noeud. On considère que le parent de la racine est un pointeur sur `NULL`

(b) Modifiez vos fonctions pour que le lien entre les noeuds parents soient correctement intégrés en cas d'ajout ou de suppression de noeud.

(c) Modifiez enfin la fonction d'affichage pour que le parent de chaque noeud soit affiché à coté de la valeur du noeud (ou `null` s'il n'existe pas. Puisque la racine a un parent pointant sur `NULL`, il est nécessaire de faire un test sur l'existence du parent avant d'accéder à sa valeur.

(d) Testez vos fonctions pour vous assurer que les parents sont correctement affectés et mis à jour en cas de modification / suppression

Remarques / questions : benoit.darties@u-bourgogne.fr