Université d'Orléans — L3 Info / PASD
Informatique — 2017–18

Travaux Dirigés sur Machine n°4 — Arbres généralisés — TD non noté (3h)

L'objectif de ce TD est de construire des arbres généralisés (nombre de fils non borné) générique (type des valeurs inconnu). L'application utilise ces arbres pour stocker et manipuler un document XML.

Le TD est divisé en deux parties. La *première partie* consiste à mettre en place les structures `noeud`, `arbre` et `arbre_parcours` et les fonctions associées. Un programme principal est disponible pour les arbres dont les nœuds sont des entiers. La *deuxième partie* consiste à travailler sur des documents XML. Les nœuds représentent alors des balises.

Il *est nécessaire* de récupérer le code *complet et fonctionnel* du module `chaine` réalisé lors du TD 1.

Exercice 1. La structure `arbre`

Pour construire un arbre généralisé, on utilise trois structures définies dans les fichiers `arbres.h` et `arbres.c` de l'archive `TD4.tgz`. La documentation (pour ce qui est visible) et des commentaires sont disponibles dans les fichiers. Leur implantation est partiellement fournie et *doit être terminée*.

La première structure permet de représenter les noeuds des arbres et est complètement cachée dans `arbre.c` . Les figures 1 et 2 illustrent les différents champs.

```
typedef struct noeud_struct* noeud;
struct noeud_struct {
  void * val;
  noeud pere;         // nœud père
  noeud fils_gauche; // fils le plus à gauche
  noeud frere_droit; // frère immédiatement à droite
} ;
```

À cette structure s'ajoute la suivante accessible depuis l'extérieur :

```
typedef struct arbre_struct * arbre; // dans arbre.h
struct arbre_struct { // dans arbre.c
  noeud racine;  // pointe sur la racine de l'arbre (le noeud qui n'a pas de père)
  void (*copier)(void* val, void** pt);
  void (*detruire) (void** pt);
};
```

Pour se déplacer dans l'arbre, il existe un troisième type `arbre_parcours`. Le détail de la structure est caché, il contient l'arbre considéré et le nœud courant. Les fonctions booléennes `arbre_parcours_est_fini`, `arbre_parcours_suivant` et `arbre_parcours_valeur` permettent de parcourir les valeurs dans un ordre préfixe.

(1) Proposer dans le *compte rendu* un autre code pour la fonction `arbre_afficher` qui n'utilise pas `noeud_afficher` et qui utilise un `arbre_parcours`.

Les fonctions `arbre_parcours_aller_fils_gauche` , `..._aller_fils_droit` , `..._aller_frere_droit` et `..._aller_pere` permettent de déplacer le nœud courant du parcours dans l'arbre. Il existe également des fonctions pour interroger un parcours ; par exemple `arbre_parcours_a_fils` permet de savoir si le nœud courant a un fils.

Une fonction importante à coder est l'extraction (`arbre_extraction`). Elle prend un arbre et une valeur recherchée en argument. Elle renvoie un arbre dont la racine est le `noeud` contenant cette valeur. Si cette valeur apparaît plusieurs fois, on choisit le nœud où on la rencontre pour la première fois dans l'ordre préfixe (cf. `noeud_rechercher`). De plus l'arbre en paramètre est également modifié : il ne contient plus le sous arbre extrait. Des exemples sont donnés sur les figures 1 et 3. Si la valeur n'apparaît pas dans l'arbre, on renvoie `NULL`.

Une autre fonction importante est la fonction `arbre_parcourir`. Elle fait subir un même traitement à toutes les valeurs contenues dans l'arbre (dans l'ordre préfixe). Pour cela elle a besoin d'un pointeur de fonction (le traitement à effectuer), `faire`, et de potentiels arguments.

(2) Expliquer *dans le compte-rendu* le type de `faire`.

Le programme `test_arbres_int.c` fournit la construction d'un arbre dont les nœuds contiennent des entiers.

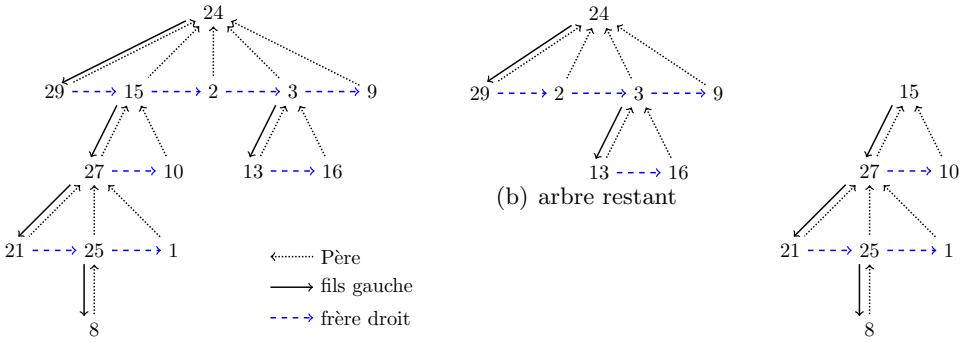
Partant de l'arbre représenté par la Fig. 1(a), l'affichage doit être :

```
24 : 29 : 15 : 27 : 21 : 25 : 8 : 1 : 10 : 2 : 3 : 13 : 16 : 9 ,
```

et pour l'affichage sous forme de tuple :

```
24 ( 29 , 15 ( 27 ( 21 , 25 ( 8 ) , 1 ) , 10 ) , 2 , 3 ( 13 , 16 ) , 9 ) .
```

Si le test d'extraction se fait sur la valeur 15, on extrait l'arbre `15 ( 27 ( 21 , 25 ( 8 ) , 1 ) , 10 )` (Figure 1(c)) et l'arbre initial est réduit à `24 ( 29 , 2 , 3 ( 13 , 16 ) , 9 )` (Figure 1(b)).



(a) arbre initial (b) arbre restant (c) arbre extrait

FIGURE 1 – Exemple d'arbre d'entiers et d'extraction.

Exercice 2. Application aux documents XML

On considère une version simplifiée de documents XML avec seulement des balises ouvrantes et fermantes. Ce sont donc des documents XML du type :

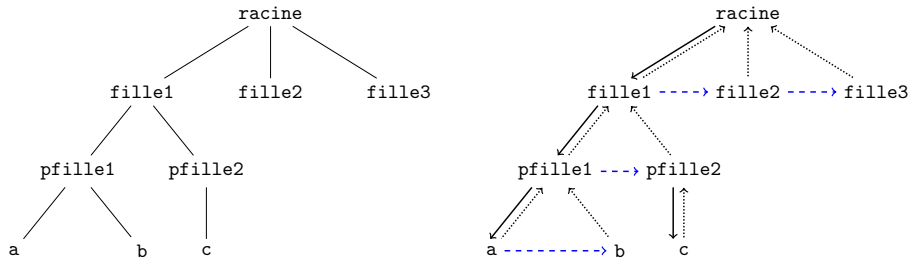
```
<racine>
  <fille1>
    <pfille1>
      <a>
      </a>
      <b>
      </b>
    </pfille1>
    <pfille2>
      <c>
      </c>
    </pfille2>
  </fille1>
   ....
</racine>
```

La structure `arbre` permet de représenter des documents XML en stockant les balises dans les nœuds comme illustré par la Figure 2.

Pour stocker le libellé des balises, on utilise le type `chaine` implanté lors du premier TD. La structure permettant de stocker une balise est :

```
typedef struct balise_struct * balise ;
struct balise_struct {
  enum { ouvrante , fermante } type;
  chaine nom ;                          //  (cf. TD1)
};
```

(3) Dessiner *dans le compte-rendu* la structure `balise_struct` complète correspondant à `</body>`.



(a) vu comme un arbre (b) structure de données

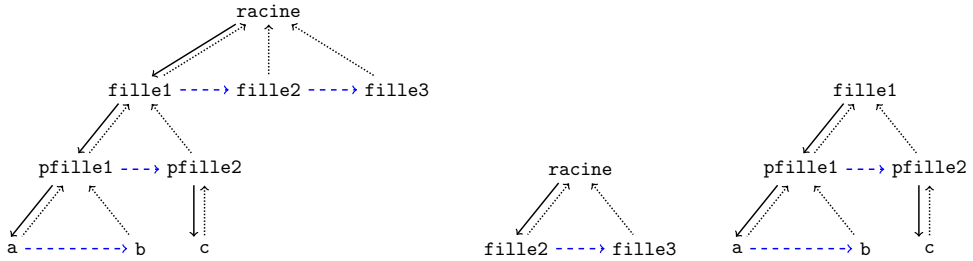
FIGURE 2 – Arbre associé à un document XML et chaînage.

Les fichiers `xml.h` et `xml.c` contiennent les déclarations et les squelettes correspondant à l'implantation de cette structure et à la lecture d'un fichier XML.

La fonction `balise_creer` prend en paramètre une chaîne de caractères qui doit être une balise (`<xxx>` ou `</xxx>`) et la transforme en ne gardant que le nom de la balise (`xxx`) et en indiquant son type.

La fonction importante à implémenter est la fonction `xml_construction_arbre` qui, à partir d'un fichier contenant un document XML simplifié, construit l'arbre correspondant. Le document XML à lire sera toujours supposé valide (la vérification n'est donc pas à faire).

La Figure 3 donne un autre exemple du principe d'extraction où `AA` est l'arbre initial et `AAA` l'arbre extrait à partir de la balise `fille1`.



(a) avant (b) après (c) extrait

FIGURE 3 – Illustration d'extraction.

Dans `TD4.tgz`, un `Makefile` est disponible avec les commandes habituelles :

— `test_arbres_int` et `test_arbres_xml` compile les deux programmes sur les entiers et les documents XML respectivement,

— `test_int` exécute l'exemple sur les entiers et `memoire_int` avec le test de la mémoire, et

— `test_xml` exécute l'exemple sur les documents XML et `memoire_xml` avec le test de la mémoire.