Année 2018-2019 Session 1

ATELIER ALGORITHMIQUE ET PROGRAMMATION
PG116

Filière : Informatique, Année : 1

Date de l'examen : 22/02/2019 Durée de l'examen : 1h20

Sujet de : T. Castanet, G. Eyrolles, F. Herbreteau, D. Renault

Documents ☐ autorisés ☒ non autorisés Calculatrice ☐ autorisée ☒ non autorisée

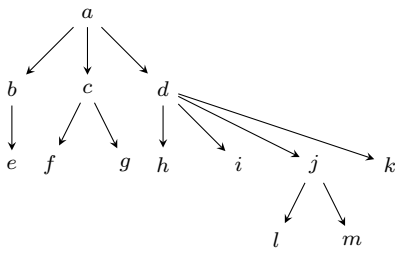
Les exercices peuvent être traités indépendamment les uns des autres.

Ce sujet porte sur une représentation et des algorithmes traitant d'arbres n-aires. Un arbre n-aire est un arbre dont chaque nœud peut avoir entre 0 et n successeurs. On peut représenter efficacement la liste des fils d'un nœud en encodant l'arbre n-aire avec des nœuds à deux successeurs.

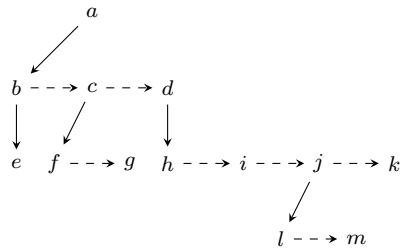
— Le premier successeur d'un nœud n est le premier fils du nœud.
— Le deuxième successeur de n est son frère suivant dans la liste des fils du nœud père de n.

La figure ci-dessous illustre cette représentation. À gauche, un arbre 4-aire, à droite, sa représentation par un arbre de nœuds à deux successeurs. En trait plein, le premier fils. En trait interrompu, le frère suivant. Par exemple, les fils du nœud a s'obtiennent en suivant le premier fils de a : $a \to b$. Puis à partir de b, la liste des frères de b : $b \dashrightarrow c \dashrightarrow d$.

On remarque que le nœud racine a n'a pas de frère. De même, le dernier fils (par exemple d) n'a pas de frère. Une feuille de l'arbre peut avoir un frère mais pas de fils (par exemple l).



Un arbre n-aire.



Sa représentation par des nœuds à 2 successeurs.

1 Mise en œuvre

Le fichier `node.h` décrit un type abstrait de données pour les nœuds à 2 successeurs. Chaque nœud stocke une donnée de type `int`.

Exercice 1 (3 points)

Dans cet exercice on s'intéresse pas aux arbres, mais uniquement aux nœuds à 2 successeurs.

Mettre en œuvre les fonctions `node__initialize`, `node__link` et `node__unlink` déclarées dans le fichier en-tête `node.h`. ♦

L'objectif des exercices suivants est de mettre en œuvre un type abstrait de donnée `ntree`, pour les arbres n-aires, utilisant le type **`struct`** `node_t` de l'exercice précédent et défini dans le fichier `node.h`.

Exercice 2 (3 points)

Mettre en œuvre les fonctions `ntree__empty` et `ntree__free` déclarées dans le fichier `ex2/ntree.h` en utilisant **`struct`** `ntree_t` et le type **`enum`** `direction_t`. Vos fonctions ne doivent pas accéder directement aux champs de **`struct`** `node_t`, mais utiliser les fonctions déclarées dans `node.h`.

La fonction `ntree__free` doit libérer toute la mémoire allouée, y compris les nœuds de l'arbre qui ne sont pas vides (indication : utiliser un algorithme récursif). ♦

La représentation des arbres n-aires par des nœuds à 2 successeurs est un choix d'implémentation qui ne doit pas être visible des utilisateurs. Le fichier `children.h` déclare le type **`struct`** `children_t` et des fonctions qui permettent de parcourir la liste des fils d'un nœud sans se soucier de sa représentation.

Exercice 3 (4 points)

Mettre en œuvre les fonctions `children__current`, `children__next` et `children__is_empty` déclarées dans le fichier `children.h`, ainsi que la fonction `ntree__children` déclarées dans le fichier `ex3/ntree.h`. ♦

Exercice 4 (4 points)

Mettre en œuvre les fonctions `ntree__degree` et `ntree__depth` déclarées dans le fichier `ex4/ntree.h`, en utilisant les fonctions de parcours `children__*` du type **`struct`** `children_t` (indication : utiliser un algorithme récursif pour `ntree__depth`). ♦

Exercice 5 (4 points)

Mettre en œuvre la fonctions `ntree__build` déclarée dans le fichier `ex5/ntree.h`. ♦

2 Questions de cours

Exercice 6 (4 points)

Répondre aux questions de cours ci-dessous dans le fichier `ex6.txt` :

1. Donner le prototype de la fonction `ntree__find` qui retourne le premier nœud d'un arbre vérifiant un prédicat.
2. Nous souhaitons stocker des valeurs de type quelconque dans les nœuds de l'arbre n-aire. Donner les modifications à apporter aux structures de données et aux prototypes de fonctions.

♦

Annexes

A.1 node.h

```
#ifndef NODE_H
#define NODE_H

struct node_t {
  int _data;
  struct node_t * _children[2];
};

/* RETURN an empty node
 */
struct node_t * node__empty();

/* PARAM n : a node
 * PARAM x : a data
 * PRECOND n is not empty
 * POSTCOND n has been initialized with data x and empty child nodes
 */
void node__initialize(struct node_t * n, int x);

/* PARAM n : a node
 * RETURN 0 if n is not an empty node, non-0 otherwise
 */
int node__is_empty(struct node_t const * n);

/* PARAM n : a node
 * PRECOND n is not an empty node
 * RETURN the data stored in node n
 */
int node__data(struct node_t const * n);

/* PARAM n : a node
 * PARAM d : a direction (0 or 1)
 * PRECOND n is not an empty node, and d is either 0 or 1
 * RETURN the child of node n at direction d
 */
struct node_t * node__child(struct node_t const * n, unsigned d);

/* PARAM n1 : a node
 * PARAM n2 : a node
 * PARAM d : a direction (0 or 1)
 * PRECOND : n1 is not an empty node, and d is either 0 or 1
 * POSTCOND : n2 is the child of n1 at direction d
 * RETURN the previous child of n1 at direction d
 */
struct node_t * node__link(struct node_t * n1, struct node_t * n2, unsigned d);

/* PARAM n : a node
 * PARAM d : a direction (0 or 1)
 * PRECOND : n is not an empty node, and d is either 0 or 1
 * POSTCOND : the child of n at direction d is an empty node
 * RETURN the previous child of n at direction d
 */
struct node_t * node__unlink(struct node_t * n, unsigned d);

#endif //  NODE_H
```

A.2 ntree.h

```
#ifndef NTREE_H
#define NTREE_H

#include "node.h"

struct ntree_t;

struct children_t {
  struct node_t * _current;
};

/* RETURN an empty n-tree
 */
struct ntree_t * ntree__empty();

/* PARAM t : a n-tree
```

```
 * PRECOND t is not NULL
 * RETURN 0 if t is not an empty n-tree, another value otherwise
 */
int ntree__is_empty(struct ntree_t const * t);

/* PARAM t : a n-tree
 * PRECOND t is not NULL
 * POSTCOND the memory allocated for the tree t and all its subtree
 * has been freed
 */
void ntree__free(struct ntree_t * t);

/* PARAM t : a n-tree
 * PRECOND t is not NULL
 * RETURN the root node of t
 */
struct node_t * ntree__root(struct ntree_t * t);

/* PARAM n : a node
 * PRECOND n is not NULL, and n is not empty
 * RETURN the sequence of child nodes of n
 */
struct children_t ntree__children(struct node_t * n);

/* PARAM children : a sequence of child nodes
 * PRECOND children is not NULL, and children is not empty
 * RETURN the node pointed by children
 */
struct node_t * children__current(struct children_t const * children);

/* PARAM children : a sequence of child nodes
 * PRECOND children is not NULL
 * RETURN 0 if children is an empty sequence of nodes, another value
 * otherwise
 */
int children__is_empty(struct children_t const * children);

/* PARAM children : a sequence of child nodes
 * PRECOND children is not NULL, and children is not empty
 * POSTCOND either children is empty, or children points to the next node
 * in the sequence
 */
void children__next(struct children_t * children);

/* PARAM n : a node
 * PRECOND n is not NULL
 * RETURN the degree of node n
 */
size_t ntree__degree(struct node_t * n);

/* PARAM t : a n-tree
 * PRECOND t is not NULL
 * RETURN the depth (height) of tree t
 */
size_t ntree__depth(struct ntree_t * t);

/* PARAM x : a value
 * PARAM nodes : an array of nodes
 * PARAM size : the size of nodes
 * PRECOND the nodes have no sibling
 * RETURN the tree with root x and children nodes in the same order
 * as in nodes
 */
struct ntree_t * ntree__build(int x, struct node_t * nodes[], size_t size);

#endif // NTREE_H
```