CONCOURS
CPGE
EPITA - IPSA - ESME
1 SEUL ET UNIQUE CONCOURS
POUR VOUS PRÉPARER
À UNE MULTITUDE
DE MÉTIERS

Samedi 8 avril 2023

OPTION : SCIENCES DU NUMÉRIQUE

MPI

Durée : 2 heures

Conditions particulières

Calculatrice interdite
Indiquez uniquement votre code candidat SCEI sur le QCM à insérer dans votre copie d'examen

Concours CPGE EPITA-IPSA-ESME 2023
Option Sciences du Numérique
Filière MPI

Consignes générales

Tout code doit être écrit dans les langages Ocaml ou C, suivant l'énoncé.

- Veuillez indenter votre code correctement.
- Tout ce dont vous avez besoin (fonctions, méthodes) est indiqué ci-dessous.
- Vous pouvez écrire vos propres fonctions, dans ce cas elles doivent être documentées (on doit savoir ce qu'elles font). Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.

Consignes Ocaml

La partie I demande d'écrire du code en Ocaml. Toute fonction du module `List` est utilisable, si nécessaire.

Consignes C

La partie II demande d'écrire du code en langage C. On supposera la bibliothèque standard (`stdlib`) et la bibliothèque `stdbool` incluses. On rappelle que cette dernière bibliothèque définit les deux booléens `true` et `false`.

Introduction

Le sujet de l'option Sciences du numérique contient trois sections. Elles sont indépendantes. Le candidat est invité à parcourir intégralement le sujet avant de commencer la rédaction.

I. Algorithmique sur les arbres et programmation en langage Ocaml : Mots de Dyck et arbres binaires

Définition 1. *On appelle mot de Dyck un mot m sur l'alphabet $\{a, b\}$ tel que :*

- *m contient autant de a que de b ;*
- *tout préfixe m contient plus de a que de b ;*

Par exemple, $m = aababb$ est un mot de Dyck, car il possède trois a et trois b, et ses préfixes sont ε (le mot vide), a, aa, aab, $aaba$, $aabab$ et m, et aucun ne contient strictement plus de b que de a.

Question 1. Parmi les mots suivants, indiquer ceux qui sont de Dyck. On ne demande pas de preuve.

$$\varepsilon \qquad aabbbaab \qquad aaab \qquad abab \qquad aaabbb$$

On représente dans la suite en Ocaml un mot sur l'alphabet $\{a, b\}$ par la liste de ses lettres. On définit les types suivants :

```
type lettre = A | B ;;
type mot = lettre list ;;
```

Le mot $m = aababb$ sera donc représenté par la liste `[A; A; B; A; B; B]`.

Question 2. Écrire une fonction `verifie_dyck` de type `mot -> bool` renvoyant un booléen indiquant si le mot passé en entrée est de Dyck.

On admet la propriété suivante : tout mot m de Dyck non vide se décompose de manière unique sous la forme $m = aubv$, où u et v sont des mots de Dyck. On pourra utiliser sans démonstration la caractérisation suivante : aub est le plus petit préfixe non vide de m contenant autant de a que de b.

Question 3. Donner sans démonstration la décomposition de chacun des mots de Dyck suivants :

$$ab \qquad aababb \qquad abababab \qquad aabbab$$

Question 4. Écrire une fonction `decompo_dyck` de type `mot -> mot * mot`, prenant en entrée un mot m supposé non vide et de Dyck (il est inutile de le vérifier), et renvoyant le couple de mots de Dyck (u, v) tel que $m = aubv$.

Il existe une bijection naturelle entre les arbres binaires stricts (tout nœud de l'arbre possède 0 ou 2 fils) et les mots de Dyck, basée sur la décomposition précédente :

- au mot vide est associé l'arbre réduit à une feuille ;
- à un mot de Dyck non vide $aubv$ où u et v sont de Dyck, on associe l'arbre binaire où le sous-arbre gauche est associé au mot u et le sous-arbre droit au mot v.

Question 5. Dessiner l'arbre associé au mot de Dyck $m = aababb$. Les nœuds ne portent pas d'étiquettes.

On définit le type suivant :

```
type arbre = F | N of arbre * arbre ;;
```

Question 6. Écrire une fonction `mot_a_arbre` de type `mot -> arbre` renvoyant l'arbre binaire strict associé à un mot de Dyck (on ne vérifiera pas que le mot passé en entrée est bien de Dyck).

Question 7. Écrire une fonction `arbre_a_mot` de type `arbre -> mot` faisant l'inverse.

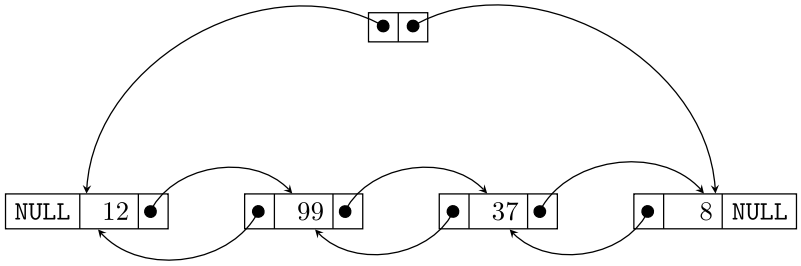
II. Algorithmique et programmation en langage C : structure de liste doublement chaînée et applications

II.1. Structure de liste doublement chaînée

On souhaite réaliser une structure de liste doublement chaînée d'entiers, permettant le retrait et l'ajout des deux côtés. Pour réaliser une telle structure, on utilise, comme sur la figure ci-dessous :

- un type `cellule` à trois champs : une valeur (entière), et deux pointeurs vers les cellules situées à sa gauche et à sa droite si elles existent, `NULL` sinon ;

— un type `dbliste` à deux champs, contenant deux pointeurs vers les cellules situées tout à gauche et tout à droite.



On définit donc les structures suivantes :

```
typedef struct cellule cellule ;
struct cellule
{
  int valeur;
  cellule* suiv;
  cellule* prec;
} ;

typedef struct dbliste dbliste ;
struct dbliste
{
  cellule* gauche;
  cellule* droite;
} ;
```

On prendra garde au fait que :

— une liste doublement chaînée ne contenant aucun élément sera représentée par un élément de type `dbliste` pour lequel les deux pointeurs `gauche` et `droite` sont `NULL` ;

— une liste doublement chaînée contenant un seul élément sera représentée par un élément de type `dbliste` pour lequel les deux pointeurs `gauche` et `droite` pointent vers la même cellule.

On définit également la fonction d'initialisation suivante :

```
void init(dbliste* q)
{
  q->gauche = NULL ;
  q->droite = NULL ;
}
```

Question 8. Écrire la fonction d'en tête `int elem_g(dbliste* q)` permettant de renvoyer l'entier situé à gauche de la liste doublement chaînée pointée par `q`, en supposant celle-ci non vide.

Question 9. Expliquer, à l'aide de schémas, le principe de l'ajout à gauche d'un entier dans la liste doublement chaînée. On fera attention à distinguer le cas d'une liste initialement vide ou non.

Question 10. Écrire la fonction d'en-tête `void push_g(dbliste* q, int v)` permettant l'ajout de l'entier `v` à gauche de la liste doublement chaînée pointée par `q`. On rappelle que `malloc(sizeof(cellule))` permet de réserver un espace mémoire de la taille d'une cellule.

On suppose dans la suite avoir écrit les fonctions d'en-tête suivantes :

— `bool vide(dbliste* q)` : renvoyant un booléen indiquant si `q` pointe sur une liste doublement chaînée vide ou non ;

— `void push_g (dbliste* q, int v)` (resp. `void push_d (dbliste* q, int v)`) permettant l'ajout de l'entier `v` à gauche (resp. à droite) de la liste doublement chaînée pointée par `q` ;

— `int elem_g(dbliste* q)` (resp. `int elem_d(dbliste* q)`) renvoyant l'entier situé à gauche (resp. à droite) de la liste doublement chaînée pointée par `q`, en supposant celle-ci non vide ;

— `int take_g(dbliste* q)` (resp. `int take_d(dbliste* q)`) supprimant et renvoyant l'entier situé à gauche (resp. à droite) de la liste doublement chaînée pointée par `q`, en supposant celle-ci non vide. La mémoire rendue disponible est libérée.