

TP n° 4: Structures de données élémentaires et applications

BATOG Guillaume

18 janvier 2007

Au cours de l'élaboration d'un algorithme, il est nécessaire de réfléchir à la manière de stocker un ensemble d'objets sur lequel on travaille pour pouvoir y opérer efficacement. Une structure de données est le pendant de la notion mathématique d'ensemble avec une spécification des opérations sous-jacentes à cette structure : création, suppression, mises à jour, requêtes,... Les listes sont un premier exemple de structure de données pouvant être définies en toute généralité de la façon suivante :

- `liste_vide ()` est un constructeur de liste vide en $O(1)$,
- `insere(x,l)` construit la liste réunion de `x` des éléments de `l` avec `x` en tête en $O(1)$,
- `supprime(x,l)` supprime l'élément `x` dans `l` en $O(|l|)$,
- `contient(x,l)` retourne vrai si `x` est un élément de `l` en $O(|l|)$.

On dispose de ces fonctions sans savoir comment elles sont réellement implémentées mais on en connaît le coût : on utilise une structure de données comme une boîte noire dans n'importe quel algorithme.

1 Piles et mots bien parenthésés

Exercice 1 [Structure LIFO]

Implémenter en Caml une structure de pile à l'aide d'une liste puis à l'aide d'un tableau, ayant pour signature :

```
val pile_vide : unit -> 'a pile (* crée une pile vide *)
val empile : 'a -> 'a pile -> 'a pile (* empile un élément en sommet de pile *)
val est_vide : 'a pile -> bool (* teste si une pile est vide *)
val sommet : 'a pile -> 'a (* retourne l'objet en sommet de pile *)
val depile : 'a pile -> 'a pile (* renvoie la pile privée de son sommet *)
```

Exercice 2 [Évaluation d'expressions arithmétiques postfixées]

Une expression arithmétique postfixée est définie par la grammaire suivante

```
nombre ::= ..., -1, 0, 1, ...
op-unaire ::= sqrt | ^-1 | abs | ...
op-binaire ::= + | - | * | / | ...
E ::= nombre | E op-unaire | E E op-binaire
```

On représentera sous Caml une forme postfixe par une suite de lexèmes :

```
type lexeme = Valeur of int
            | Op_unaire of int -> int
            | Op_binaire of int -> int -> int;;
```

Écrire un évaluateur sous Caml qui, à une forme postfixe, retourne le résultat de l'expression arithmétique correspondante.

Exercice 3 [Mots bien parenthésés]

Le langage des mots bien parenthésés à $2n$ lettres, noté \mathcal{D}_n , est défini de la manière suivante : on considère un alphabet à n lettres $Z_n = \{a_1, \dots, a_n\}$ et l'alphabet barré correspondant $\bar{Z}_n = \{\bar{a}_1, \dots, \bar{a}_n\}$. On définit la suite d'ensemble de mots $L_{n,k}$ par $L_{n,0} = \{\epsilon\}$ et pour tout k , $L_{n,k+1} = L_{n,k} \cup L_{n,k}^2 \cup \left(\bigcup_{1 \leq i \leq n} a_i L_{n,k} \bar{a}_i \right)$. Enfin on définit $\mathcal{D}_n = \bigcup_{k \in \mathbb{N}} L_{n,k}$.

- Montrer que lorsque $\omega \in \mathcal{D}_n$, alors pour tout i tel que $\omega_i \in Z_n$, il existe $j > i$ tel que $\omega[i..j] \in \mathcal{D}_n$. Si j est minimal, on dit que ω_j est la fermante associée à l'ouvrante ω_i , et réciproquement.
- En utilisant une pile, écrire une fonction Caml `association` qui, pour un mot d'entrée w , calcule en temps linéaire un vecteur \mathbf{v} tel que $\mathbf{v}.(i) = j$ si et seulement si w_i et w_j sont associées lorsque $w \in \mathcal{D}_n$ et retourne une exception sinon.
- Quel automate fini reconnaît le langage \mathcal{D}_n ?

Pour s'entraîner : Centrale 2003.

2 Files

Exercice 4 [Structure FIFO]

On veut implémenter en Caml une structure de file ayant pour signature :

```
val file_vide : unit -> 'a pile (* crée une file vide *)
val enqueue : 'a -> 'a file -> 'a file (* enqueue un élément en queue de file *)
val is_empty : 'a file -> bool (* teste si une pile est vide *)
val dequeue : 'a file -> 'a * 'a file (* retourne l'objet en sommet de file et sa queue *)
```

- en utilisant un tableau. On construira le type produit suivant :

```
type 'a file = { contenu : 'a vect ; mutable vide : bool ; mutable objet : 'a ;
                mutable debut : int ; mutable fin : int }
```
- en utilisant un couple de listes : la première liste est constituée des éléments les plus récents de la file, classés suivant leur ordre d'arrivée ; la seconde liste est constituée des éléments les plus anciens de la file, conservés dans l'ordre inverse d'arrivée. Par exemple, la file 1,2,3,4,5 (1 enfilé en premier) peut être représentée par $([1;2;3], [5,4])$ ou $([], [5;4;3;2;1])$. Dans le cas où la première liste est vide, on effectuera une opération de normalisation : $([], l)$ devient $(\tilde{l}, [])$.

Exercice 5 [Parcours en largeur d'un graphe]

Un graphe orienté $\mathcal{G} = (V, E)$ est la donnée d'un ensemble de sommets V et d'un ensemble d'arêtes $E \subseteq V \times V$. Dans cet exercice, les sommets seront numérotés de 0 à $n - 1$ et E sera représenté par un tableau `succ` de taille n où `succ.(i) = {j ∈ V | (i, j) ∈ E}`. Un parcours en largeur de \mathcal{G} à partir d'un sommet v consiste à rencontrer successivement les sommets à distance 0, puis 1, puis 2, ... de v jusqu'à obtenir tous les sommets de \mathcal{G} accessibles à partir de v .

Construire une fonction récursive `parcoursLargeur` qui, étant donnés un graphe \mathcal{G} et un sommet initial v_0 , retourne l'ensemble des sommets rencontrés lors d'un parcours en largeur de \mathcal{G} à partir de v_0 .

- les sommets à visiter seront stockés dans une file ;
- on maintiendra un tableau de booléens de taille n indiquant si un sommet a déjà été traité ;
- supplément : on peut retourner une liste des sommets dans l'ordre de leur visite ;
- supplément : on peut maintenir un tableau `pre` de taille n qui indique, pour un sommet visité i , le sommet j à partir duquel (en un pas) i a été visité pour la première fois. Application : trouver le chemin le plus court permettant de sortir du labyrinthe.