TD6: Structures de données

MP2I Lycée Pierre de Fermat

Exercice 1.

Notation Polonaise Inversée

On s'intéresse dans cet exercice à la ***notation polonaise inverse*** (**NPI**), ou ***notation postfixe***, qui est une manière de noter les expressions arithmétiques. On considère une expression arithmétique avec les notations habituelles :

$$(a+b)\times(c-(d-2))$$

Dans une telle expression, on appellera les $a, b, ...$ et les nombres des ***constantes***.

On dit que cette expression utilise la notation ***infixe*** car les opérateurs sont entre les opérandes (ex : dans $a+b$, le $+$ est entre le a et le b). Le principe de la **NPI** est d'écrire les opérateurs après les opérandes. Ainsi, pour effectuer la somme de a et b, on écrira :

$$b\ a\ +$$

et pour écrire $a-b$ on écrira :

$$b\ a\ -$$

Remarquons que l'ordre des opérandes est inversé Ainsi, l'expression donnée plus haut s'écrit en **NPI** :

$$((2\ d\ -)\ c\ -)\ (b\ a\ +)\ \times$$

Nous allons voir que les parenthèses sont inutiles, car il existe une unique manière de lire une expression **NPI**. On écrira donc :

$$2\ d\ -\ c\ -\ b\ a\ +\ \times$$

Pour lire une expression arithmétique écrite en **NPI**, on utilise une pile :

— Initialement, la pile est vide, et on démarre à gauche de l'expression

— On lit chaque opérateur et constante dans l'ordre :

  — Lorsque l'on lit une constante, on l'empile

  — Lorsque l'on lit un opérateur, on dépile deux éléments de la pile, on leur applique l'opérateur et on empile le résultat. Le premier argument de l'opérateur est le premier élément dépilé. Par exemple, si l'on fait une soustraction et que la pile contient 1 au sommet et x en dessous, l'opération sera $1-x$.

— Si l'expression n'a pas d'erreur de syntaxe, la pile contiendra un unique élément à la fin : c'est la valeur de l'expression !

Q1. Appliquer la procédure décrite sur l'expression donnée plus haut, et vérifier que l'on obtient bien la même valeur qu'en notation infixe.

Q2. Les expressions suivantes sont-elles valides en **NPI** ? Si oui, donner leur traduction en notation infixe :

(a) $a\ 2\ c\ d\ -\times+4\ b-/$
(b) $x+y$
(c) $x\ y+z$
(d) $x\ y+z\times$

Q3. Traduire les expressions suivantes en **NPI** :

(a) $a\times c+b$
(b) $a\times(c+b)$
(c) $2\times(a\times a\times a+1)$
(d) $(1-x)\times(2+(x/(1-x))+x)$

Q4. Dans une expression en **NPI**, donner une relation entre N_{op} le nombre d'opérateurs et N_c le nombre de constantes. Toute expression vérifiant cette relation est elle une expression correcte en **NPI** ?

On rajoute maintenant à nos opérateurs les fonctions $(\sin,\cos,\log,\dots)$. Lorsque l'on lit un nom de fonction, on dépile un élément, on lui applique la fonction, et on empile le résultat. Par exemple, pour écrire $\sin(\frac{3\pi}{4})$ en **NPI** :

$$4\ 3\ \pi\ \times\ /\ \sin$$

On introduit la notion d'***arité*** d'un opérateur : l'arité est le nombre d'opérandes que prend l'opérateur. Par exemple, $+$ et $\times$ sont d'arité 2, et sin est d'arité 1. On note $a(o)$ l'arité d'un opérateur o. De plus, **on considère les constantes comme des opérateurs d'arité 0**.

Q5. On considère une expression e en NPI. On note $o_1\dots o_n$ ses opérateurs (constantes incluses). Proposer une relation généralisant la propriété trouvé à la question précédente.

On rajoute deux nouveaux opérateurs, noté **DUP** et **SWAP**, exclusifs à la **NPI**.

— **DUP** sert à dupliquer son opérande : lorsqu'on lit **DUP**, on dépile le sommet de la pile, et en empile deux copies. Par exemple :

$$x\ 1+\textbf{DUP}\ \times$$

va s'évaluer à $(x+1)\times(x+1)=(x+1)^2$

— **SWAP** sert à échanger ses deux opérandes : lorsqu'on lit **SWAP**, on dépile deux éléments de la pile, et les empile dans l'ordre inverse. Par exemple :

$$1\ x\ \textbf{SWAP}\ /$$

va s'évaluer à $\frac{1}{x}$

Q6. Généraliser à nouveau la relation des questions 4 et 5.

Q7. Écrire en **NPI** l'expression suivante, en utilisant le moins de symboles (constantes + opérateurs) possible :

$$\frac{1+\sin(\frac{3\pi}{4})}{3+\frac{3\pi}{4}}$$

Exercice 2.

Opérations de pile

Q1. Que fait l'algorithme suivant ?

Entrée(s) : P une pile
Sortie(s) : ? ? ?

1 $Q \leftarrow$ **pile_vide**();
2 **tant que** P ***n'est pas vide*** **faire**
3 $\quad x \leftarrow$ **dépiler**(P);
4 $\quad$ **empiler**(Q, x) ;
5 **retourner** $\boldsymbol{Q}$

Q2. En s'inspirant de l'algorithme précédent, écrire un algorithme permettant de créer une copie d'une pile P. La pile en entrée pourra être modifiée pendant l'exécution, mais à la fin de l'algorithme, elle devra être intacte.

Q3. Écrire un algorithme permettant de calculer la taille d'une pile, i.e. le nombre d'éléments qu'elle contient.

Q4. Donner un algorithme permettant de tester l'égalité entre deux piles. Cet algorithme renverra vrai lorsque les deux piles en entrée contiennent les mêmes éléments dans le même ordre, faux sinon. Les piles ne doivent pas être modifiées en sortie.

On souhaite maintenant rajouter aux piles l'opération **taille**, en tant qu'opération de base. Une première possibilité est de reprendre l'algorithme de la Q3 en $\mathcal{O}(|P|)$, mais on peut en fait trouver des solutions plus efficaces si l'on revient aux implémentations concrètes.

Q5. Comment réaliser cette opération en $\mathcal{O}(1)$ dans l'implémentation par tableaux ?

Q6. On considère l'implémentation par listes chaînées des piles vue en cours :

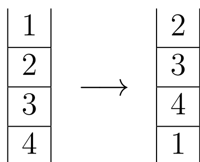
```c
struct maillon {
  int donnees;
  struct maillon* suivant;
};
typedef struct maillon maillon_t;

struct liste {
  maillon_t* tete;
};
typedef struct liste liste_t;
```

Écrire une fonction `int taille(liste_t* L)` s'exécutant en $\mathcal{O}(n)$ calculant la taille d'une pile.

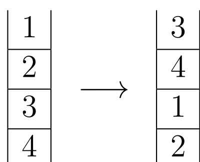
Q7. Comment peut-on modifier l'implémentation par listes chaînées pour pouvoir obtenir la taille en $\mathcal{O}(1)$?

Exercice 3. *Rotation de pile*

Une ***rotation unitaire*** de pile consiste à prendre l'élément au sommet de la pile, et à le déplacer à la base. Voici un exemple de rotation unitaire :



Une ***rotation d'ordre*** $k \in \mathbb{N}$ est une succession de k rotations unitaires. Voici un exemple de rotation d'ordre 2 :



Q1. Écrire un algorithme **Rotation-Unitaire** permettant d'effectuer une rotation unitaire sur une pile P. Sa complexité devra être en $\mathcal{O}(|P|)$.

Q2. Décrire en une phrase un algorithme permettant d'effectuer $k \in \mathbb{N}$ rotations sur une pile P, en temps $\mathcal{O}(k \times |P|)$.

Q3. Écrire un algorithme **Rotation** permettant d'effectuer $k \in \mathbb{N}$ rotations sur une pile P. Sa complexité devra être en $\mathcal{O}(|P|)$.

Exercice 4. *Génération de mots bien parenthésés*

Nous avons vu en cours un algorithme permettant de déterminer si un mot est bien parenthésé, en utilisant une pile. On s'intéresse ici à la **génération** de mots bien parenthésés, d'une manière plus efficace que de générer un mot au hasard jusqu'à ce qu'il soit bien parenthésé.

Plus précisément, étant donné un entier $n \in \mathbb{N}$, on souhaite générer un mot de taille $2n$ bien parenthésé sur l'alphabet `(, ), [, ]`.

Le principe va être d'utiliser une pile pour stocker les parenthèses ouvertes n'ayant pas encore été fermées. A chaque étape de l'algorithme, on va tirer à pile ou face, et selon le résultat :

— soit ajouter une parenthèse ouvrante au mot, choisie aléatoirement parmis les parenthèses ouvrantes, et donc empiler la parenthèse sur la pile ;

— soit ajouter une parenthèse fermante au mot, auquel cas ce doit être la parenthèse correspondante du sommet de pile.

On garde également en mémoire le nombre de parenthèses ouvrantes, de façon à ce que l'algorithme en ouvre exactement n.

Q1. Appliquer l'algorithme pour générer quelques mots bien parenthésés

Q2. Écrire le pseudo-code correspondant.

Exercice 5.

Modélisation

On souhaite modéliser une usine de fabriquation de burgers à la chaîne. Les burgers fabriqués se déplacent sur des tapis roulants, en étant déplacés de point en point. A chaque point, des machines ajoutent au burger un ingrédient. Les burgers démarrent vides, avec juste un pain inférieur, au point de départ. Une fois au point d'arrivée, les burgers sont complétés par un pain supérieur, emballés et sortis de l'usine. Chaque burger créé correspond à une commande, indiquant pour chaque ingrédient si oui ou non le burger doit le contenir

On représente donc une usine par :

— Des points $P_0, P_1, \ldots, P_n, P_n$, avec P_0 le point de départ et P_n le point d'arrivée.

— Des tapis roulants $T_0, \ldots, T_{n-1}$, avec T_i amenant les produits du point P_i au point P_{i+1}

— Des burgers contenant des ingrédients mis dans l'ordre des points

— Des commandes de burgers. Une commande est un sous-ensemble $C \subseteq [\![1, n]\!]$ contenant l'indice des ingrédients souhaités.

On simule ensuite le fonctionnement de l'usine en parallèle :

— Chaque point P_i prend un temps $t_i \in \mathbb{N}$ pour ajouter son ingrédient.

— Chaque tapis prend un temps considéré comme négligeable pour transporter les burgers d'un point à un autre, et peut transporter un nombre infini de burgers.

Q1. Proposer des structures à utiliser pour les ingrédients, les commandes, les burgers, les points, les tapis.

Q2. Écrire une procédure **simuler_point**(i) qui simule une étape du point i pour $i \in [\![1, n]\!]$

Q3. Écrire une procédure **simuler_usine**$()$ qui simule une étape de l'usine entière. Y a-t-il une différence entre simuler les points du début vers l'arrivée et de l'arrivée vers le début ? Quelle option semble la plus appropriée ?

Exercice 6.

File avec deux piles

On étudie une implémentation des files utilisant deux piles, une pile-queue et une pile-tête :

- On enfile les éléments en les empilant sur la pile-queue
- On défile les éléments en dépilant la pile-tête
- Si la pile-tête est vide lorsque l'on défile, on commence par **transvaser** toute la pile-queue dans la pile-tête.

Q1. On considère une file F initialement vide. Représenter F, sous forme de file abstraite et dans l'implémentation à deux piles proposée, après chacune des opérations suivantes :

`enfiler 1; enfiler 2; enfiler 3; défiler;`
`défiler; enfiler 4; défiler; enfiler 5;`
`enfiler 6; défiler; enfiler 7`

En C, on utiliserait le type suivant :

```
typedef struct file_{
  pile_t* pile_tete;
  pile_t* pile_queue;
} file_t;
```

en supposant que l'on dispose d'un type de pile `pile_t` et des 4 opérations des piles :

```
pile_t* creer_pile();
bool pile_est_vide(pile_t* p);
void empiler(pile_t* p, T x); // T est le type des données stockées
T depiler(pile_t* p);
```

Q2. Rappeler les 4 opérations de la SDA de file, et les implémenter en C avec la structure concrète à deux piles proposée. Donner la complexité asymptotique de chacune en fonction de la taille n de la file.

Q3. Supposons qu'une opération de défilage a nécessité de transvaser K éléments de la pile queue à la pile tête. Que peut-on dire des enfilages et des défilages ayant été effectués sur la file entre ce transvasage et le précédent ?

Q4. On considère une pile vide P, et une suite d'opérations $o_1, \dots o_n$ avec chaque o_i étant soit un enfilage, soit un défilage. Donner le coût total asymptotique des opérations $o_1, \dots, o_n$ en fonction de n.

Q5. Quel est le coût moyen d'une opération dans cette implémentation ?

La remarque précédente exprime le fait que même si **une** opération peut être coûteuse, une **suite** d'opérations sera, en moyenne, pas trop coûteuse car les opérations chères seront rares, et donc compensées par les opérations pas chères.

L'étude du coût moyenné de suites d'opérations s'appelle **l'analyse amortie**, et le coût d'une opération est alors appelé sa complexité amortie. Nous avons ainsi montré dans cet exercice que l'implémentation proposée des files permet d'enfiler et de défiler en temps constant **amorti**.

Opérations des SDA

Piles

- Créer une pile vide
- Empiler un élément au sommet
- Dépiler le sommet et le renvoyer
- Déterminer si une pile est vide

Files

- Créer une file vide
- Enfiler un élément à la queue
- Défiler l'élément de tête et le renvoyer
- Déterminer si une file est vide