Université de Paris

L2 informatique - Année 2021–2022

TD d'Éléments d'Algorithmique n° 7 (Correction)

Exercice 1. *Insertion dans une liste doublement chaînée.*
Écrivez le pseudocode d'une fonction INSERER(L, k, x) qui prend comme argument une liste doublement chaînée L, un élément non nul x de L et une clé k et insère un nouveau nœud ayant la clé k avant x.

Correction :
Entrée : une liste d.c. `L`, un pointeur `x` sur un nœud non nul de `L` et une clé `k`
Sortie : la liste dans laquelle `y` a été inséré avant `x`

```
1: fonction INSERER(L, x, y)
2:     y ← new Node(k, null, null)
3:     y.next ← x
4:     y.prev ← x.prev
5:     si x.prev ≠ null alors
6:         x.prev.next ← y
7:     sinon
8:         L.head ← y
9:     x.prev ← y
```

Rappel. Une pile est une structure de données *abstraite* sur laquelle sont définies trois opérations :

- `empty(P)` qui teste si la pile `P` est vide ;
- `push(x,P)` qui ajoute un élément `x` au sommet de la pile `P` ;
- `pop(P)` qui enlève la valeur au sommet de la pile `P` et la renvoie.

Exercice 2. *Sedgewick.*
Dans la suite suivante, une lettre indique un *push* et un astérisque indique un *pop*. Donnez la suite de valeurs retournées par les *pop* exécutés lorsqu'on applique à une pile la suite d'opérations suivante :

```
L A * S T I * N * F I R * S T * * O U * T * * * * * *
```

Correction :

```
1 AINRTSUTOIFTS
```

Exercice 3. *Réalisations concrètes.*

1. Implémentez une pile en la représentant comme une liste chaînée simple.
2. On suppose maintenant que la taille maximale de la pile est donnée lors de sa création. Implémentez une pile en la représentant par la structure suivante :

```
class Stack {
  int[] T;
  int top;
}
```

Exercice 4. *La récursion revisitée.*

1. Écrivez un algorithme itératif qui, étant donnée une liste `L`, retourne une nouvelle liste qui contient les éléments de `L` dans l'ordre inverse. Votre algorithme s'exécutera en temps et en espace linéaires (proportionnels à la longueur de `L`), et il pourra détruire la liste passée en paramètre.
2. Écrivez un algorithme récursif qui, étant donnée une liste doublement chaînée `L`, retourne une nouvelle liste qui contient les éléments de `L` dans l'ordre inverse. Vous ne devez pas utiliser de mémoire supplémentaire, sauf pour les pointeurs.
3. Écrivez un algorithme récursif qui concatène deux listes. Vous ne devrez utiliser aucun espace en plus de celui qui est utilisé pour implémenter la récursion.
4. Écrivez un algorithme itératif qui concatène deux listes. Vous pouvez détruire l'une des deux listes.

On suppose maintenant données des opérations `read` et `write` qui lisent et écrivent un caractère, et l'opération `tochar` qui convertit un entier en le caractère correspondant.

5. Écrivez un algorithme itératif qui lit un nombre `n` de caractères et les affiche dans l'ordre inverse.
6. Écrivez un algorithme itératif qui affiche la valeur d'un nombre en base 8.

Correction :

```
reverse(L):
    M := new List(nil)
    c := L.tete
    while c != nil {
        push(c.cle, M)
        c = c.suiv
    }
    return M
```

```
  rev(L) {
    M := L;
    rev_aux(M,M.head);
  }

  rev_aux(M,h) {
    if (h = null) return M;
    aux:= h.next;
    h.next:= h.prev;
    h.prev:= aux;
    M.head:= h;
    rev_aux(M, aux);
  }
```

```
append'(c, d):
    if c = nil
        return d
    else
        return new Cellule(c.cle, append'(c.suiv, d))

appendRec(L, M):
    return new List(append'(L.tete, M.tete))
```

Il y a deux solutions – on peut soit reconstruire la première liste, soit parcourir la première liste puis muter le pointeur `suiv` du dernier élément.

```
append(L, M):
    N := new List(nil)
    c := L.tete
    while c <> nil {
        push(c. cle, N)
```

```
        c := c.suiv
    }
    while not empty(N)
        push(pop(N), M)
    return M
```

```
read_reverse(n):
    L = new List(nil)
    for i := 1 to n
        push(read(), L)
    while not empty(L)
        write(pop(L))
```

```
write8(n):
    if n = 0
        print(tochar(0))
    while n != 0 {
        print(tochar(n mod 8))
        n := n div 8
    }
```

Exercice 5. *Notation polonaise inverse.*
La *notation polonaise inverse* (RPN) ou *notation postfixe* est une notation pour les expressions arithmétique qui met un opérateur après ses opérandes et pas entre eux. Par exemple, l'expression $2 + 3$ s'écrit « `2 3 +` » en RPN, et l'expression $2 \times 3 + 4$ s'écrit « `2 3 * 4 +` ».

1. Écrivez les expressions suivantes en RPN :
   (a) $1 - 2 - 3$;
   (b) $1 - (2 - 3)$;
   (c) $1 \times 2 + 3 \times 4$;
   (d) $1 - 2 \times 3 + 4$.

Une expression en RPN s'évalue très facilement à l'aide d'une seule pile [1] : un opérande correspond à un *push*, un opérateur correspond à deux *pop*, du calcul de l'opération, et d'un *push* du résultat.

2. Écrivez un algorithme qui, étant donnée une liste de nombres et d'opérandes représentant une expression en RPN, retourne la valeur de l'expression. Exécutez votre algorithme sur les expressions RPN que vous avez obtenues à la question précédente, et vérifiez qu'il produit le bon résultat.
3. Modifiez votre algorithme pour qu'il détecte les expressions incorrectes (il y a deux cas à gérer). Vous pouvez supposer donnée une fonction `error` qui termine l'exécution de l'algorithme.
4. Que faut-il changer pour ajouter un opérateur unaire de passage à l'opposé ? Peut-on utiliser le même symbole pour le passage à l'opposé et la soustraction ?
5. Écrivez une calculatrice RPN dans votre langage de programmation favori.

Correction :

1. `1 2 - 3 -`, `1 2 3 - -`, `1 2 * 3 4 * +`, `1 2 3 * - 4 +`.
2. Il faut une distinguer les opérations des symboles qui représentent les opérateurs, j'ai choisi `{*}` pour le symbole de multiplication etc. La seule difficulté est dans l'ordre des opérandes.

```
eval(L):
    S := new List(nil)
    while not empty(L) {
```

1. Il faut deux piles pour évaluer une expression en notation *infixe* habituelle.

```
4           e := pop(L)
5           if e = {+} {
6               e2 := pop(S)
7               e1 := pop(S)
8               push(e1 + e2, S)
9           else if e = {-} {
10              -- attention a l'ordre
11              e2 := pop(S)
12              e1 := pop(S)
13              push(e1 - e2, S)
14          else if e = {*} {
15              ...
16          else
17              push(e, S)
18      }
19      return pop(S)
```

3. Il faut (1) vérifier que la pile n'est pas vide quand on fait un *pop* et (2) vérifier que la pile est vide après le *pop* à la fin.
4. Rien ne change, sauf qu'on a un cas qui fait un seul *pop* (on peut aussi facilement avoir des opérateurs ternaires. Par contre, on ne peut pas surcharger les opérateurs sans ambigüité, il faut utiliser un autre symbole, par exemple `{~}` pour l'opposé unaire.