Correction de TD
Piles et Files

Ce document contient quelques éléments de correction pour les TD. Il n'est pas voué à être exhaustif et ne se substitue **pas** à la correction vue en TD.

1 Primitives

Les piles et les files peuvent vous rappeler les listes simplement chaînées : pour regarder le dernier élément, on est obligé de regarder tous ceux avant. Attention, qu'ici les primitives `depiler`/`defiler` qui permettent de parcourir une pile/file retirent les éléments : on perd donc le haut de la pile pour accéder à ceux en dessous. Pour ne pas les perdre, on peut les copier. Mais comme on le verra ensuite, copier une pile dans une autre pile inverse l'ordre des éléments.

Piles

```
pile creerPile()
bool estVide(pile P)
obj valeur(pile P)
pile depiler(pile P)
pile empiler(pile P, obj x)
```

Files

```
file creerFile()
bool estVide(file F)
obj valeur(file F)
file defiler(file F)
file enfiler(file F, obj x)
```

2 Déplacer et copier

On se donne une pile P et qu'on veut en déplacer les éléments dans une autre pile P'. Pour cela, on récupère la valeur en tête de P et on la place dans P'. Ensuite, on dépile P. Attention que `depile` supprime la valeur en tête, il faut donc la récupérer avant. Notez qu'on n'impose pas que P' soit vide. On verra avec l'exercice suivant que ce cas peut-être utile.

Algorithm 1: Deplacer

Input: P pile à déplacer dans P'

```
1 while non estVide(P) do
2     v ← valeur(P)
3     P ← depiler(P)
4     P' ← empiler(P', v)
5 return P'
```

On propose aussi une version récursive.

Algorithm 2: DeplacerRec

Input: P pile à déplacer dans P'

```
1 if estVide(P) then
2     return P'
3 else
4     return DeplacerRec(depiler(P), empiler(P', valeur(P)))
5 return P'
```

Les éléments de P sont ainsi placés en haut de la pile P', mais dans l'ordre inverse. Le dernier élément de P se retrouve donc en haut de P'. Si on veut déplacer une pile en conservant l'ordre, il faut utiliser l'algorithme précédent deux fois. Inverser l'ordre deux fois revient à l'ordre initial (on appelle cette opération une involution).

Cet algorithme est linéaire en le nombre d'élément de P, i.e. $O(n)$ si n est le nombre d'éléments dans P :

- la boucle *while* est linéaire puisque P contient n éléments au début de la boucle,
- chaque étape de la boucle est en temps constant.

On a perdu la pile P en la dépilant. Si on veut copier, il faut remplir une deuxième pile en parallèle quand on dépile.

Algorithm 3: Copier

Input: P pile

```
1 P'' ← creerPile()
2 P'' ← Deplacer(P, P'')
3 while non estVide(P'') do
4     v ← valeur(P'')
5     P'' ← depiler(P'')
6     P ← empiler(P, v)
7     P' ← empiler(P', v)
8 return P'
```

Cet algorithme renvoie une nouvelle pile P' identique à P, tout en conservant intacte la pile P. Sa complexité est linéaire aussi :

- `Deplacer` est linéaire,
- la boucle *while* est linéaire puisque P'' contient les éléments de P au début de la boucle,
- chaque étape de la boucle est en temps constant.

3 Séparer pairs et impairs

On se donne en entrée une pile d'entiers P_1. On veut copier ces éléments dans une autre pile P_2, tout en plaçant les entiers pairs en dessous des entiers impairs. La version itérative commence par séparer les pairs et les impairs dans deux piles différentes P_2 (pairs) et P_3 (impairs). Pour cela, on parcourt P_1 comme pour déplacer, sauf que cette fois, la valeur en tête est placée dans la pile

correspondante à sa parité. Une fois $P1$ séparée, on déplace les éléments impairs (dans P_3) sur la pile P_2 des éléments pairs.

Algorithm 4: Separer

```
Input: P1 une pile
P2 ← creerPile()
P3 ← creerPile()
while non estVide(P1) do
    v ← valeur(P1)
    P1 ← depiler(P1)
    if v ≡ 0[2] then
        P2 ← empiler(P2, v)
    else
        P3 ← empiler(P3, v)
Deplacer(P3, P2)
return P2
```

Les éléments de pairs n'ont été déplacés qu'une seule fois. Donc une fois dans P_2, ils sont dans l'ordre inverse de P_1. Par contre, les impairs ont été déplacés deux fois, ils finissent donc dans l'ordre initial.

On propose maintenant une version récursive.

Algorithm 5: SeparerRec

```
Input: P1, P2 deux piles
if estVide(P1) then
    return P2
else
    if valeur(P1) ≡ 0[2] then
        return separer(depiler(P1), empiler(P2, valeur(P1))
    else
        P2 = separer(depiler(P1), P2)
        P2 = empiler(P2, valeur(P1))
```

Algorithm 6: SeparerRecInit

```
Input: P1 une pile
return SeparerRec(P1, creerVide())
```

4 Bon parenthésage

On se donne une chaîne de caractère et on souhaite savoir si elle est bien parenthésée. Pour cela, il faut que toute parenthèse ouverte soit refermée, une et une seule fois. Ainsi, si on parcourt la chaîne de caractère de gauche à droite, il faut qu'à tout instant on ait vu plus (ou autant) de parenthèse ouvrante que fermante. Il faut aussi qu'à la fin du parcourt, on en ait vu autant de chaque.

On peut faire cela avec un compteur de parenthèse ouvrante non refermée. Ce compteur est :

- initialisé à 0,
- incrémenté pour chaque parenthèse ouvrante croisée,
- et décrémentée pour chaque parenthèse fermante.

Il suffit ensuite de tester que :

- le compteur est toujours positif,
- il est égal à 0 à la fin.

Algorithm 7: BonParenthesage

```
Input: s une chaîne de caractère
1  cpt ← 0
2  for i de 0 à n − 1 do
3      if s[i] =' (' then
4          cpt ← cpt + 1
5      else if s[i] =')' then
6          if cpt = 0 then
7              return i
8          else
9              cpt ← cpt − 1
10     return cpt = 0
```

Si le mot est bien parenthésé, on dit que c'est un mot de Dyck (en retirant les caractères autres que les parenthèses).

Pour simuler ce compteur, on peut utiliser une pile. Une pile vide avec i éléments représente un compteur à la valeur i. A l'instar de l'algorithme précédent, il suffit de :

- commencer avec une pile vide,
- empiler une valeur (quelconque pour l'instant) pour chaque parenthèse ouvrante,
- dépiler pour chaque parenthèse fermante.

Enfin, on vérifie que :

- on n'essaie jamais de dépiler quand la pile est vide,
- la pile est vide à la fin.

On peut adapter cette version pour obtenir plus d'information : où se situe la parenthèse fautive. Pour cela, lorsque l'on croise une parenthèse ouvrante, la valeur empilée est l'indice courant (donc la position de la parenthèse ouvrante). Si on essaie de dépiler une pile vide, on renvoie l'indice courant : c'est l'endroit où on veut fermer une parenthèse qui n'a jamais été ouverte. Sinon, si à la fin la pile n'est pas vide, on renvoie la valeur en tête : c'est l'endroit où la dernière parenthèse ouvrante non refermée a été vue. Enfin, on renvoie -1 si le parenthésage est correct.

Algorithm 8: BonParenthesagePosition

```
Input: s une chaîne de caractère
P ← creerVide()
for i de 0 à n − 1 do
    if s[i] =′ (′ then
        P ← empiler(P,′ (′)
    else if s[i] =′)′ then
        if estVide(P) then
            return i
        else
            P ← depiler(P)
    if cpt ≠ 0 then
        return valeur(P)
    else
        return −1
```

On propose une dernière version plus générale afin de mieux illustrer l'utilité de la pile. Cette fois, vous ne vous limitez plus aux parenthèses ; il y a aussi des accolades '{', '}', des crochets '[', ']', etc... On se donne les fonctions suivantes :

- `ouvrant`(c) qui renvoie Vrai si et seulement si le caractère c est une parenthèse/accolade/crochet/... ouvrant,
- `fermant`(c) qui renvoie Vrai si et seulement si le caractère c est une parenthèse/accolade/crochet/... fermant,
- `inverse`(c) qui pour un signe fermant, renvoie l'ouvrant correspondant. `inverse`('}') = '{'.

Il faut cette fois être plus vigilant : on ne peut refermer une accolade que si le dernier signe ouvrant étant aussi une accolade. Par exemple, "(abc{de)fg}" n'est pas correct ! Pour prendre cette contrainte en compte, il suffit de stocker sur la pile le caractère ouvrant. Puis on vérifie lorsque l'on croise un caractère fermant qu'il est bien du même type que celui au sommet de la pile.

Algorithm 9: BonParenthesageGeneral

```
Input: s une chaîne de caractère
P ← creerVide()
for i de 0 à n − 1 do
    if ouvrant(s[i]) then
        P ← empiler(P, s[i])
    else if fermant(s[i]) then
        if estVide(P) ou valeur(P) ≠ inverse(s[i]) then
            return i
        else
            P ← depiler(P)
    return estVide(P)
```

5 Palindrome

On se donne une chaîne de caractère s de longueur n et on cherche à vérifier si elle représente un palindrome, c'est dire si $\forall i \in [\![0; n-1]\!], s[i] = s[n-i-1]$. On va placer notre chaîne de caractère dans une pile et dans une file en parallèle en la parcourant par indices croissants. En dépilant la pile obtenue, notre chaîne de caractère est dans l'ordre inverse, tandis qu'en défilant la file, elle est dans le bon ordre. Si on prend deux valeurs de même rang dans chaque structure, ce sont deux caractères à des positions symétriques dans la chaîne de caractère.

Algorithm 10: Palindrome

```
   Input: s une chaîne de caractère
 1 P ← creerPile()
 2 F ← creerFile()
 3 for i de 0 à n − 1 do
 4 |   P ← empiler(P, s[i])
 5 |   F ← enfiler(F, s[i])
 6 while non estVide(P) do
 7 |   if valeur(P) ≠ valeur(F) then
 8 |   |   return false
 9 |   P ← depiler(P)
10 |   F ← depiler(F)
11 return true
```

La complexité est linéaire puisqu'on copie une chaîne de caractère dans une pile et une file (donc une boucle à n étapes dont chaque étape est en $O(1)$). Ensuite on dépile et on dépile deux structures de taille n, ce qui ajoute une complexité linéaire.

6 Faire des files avec des piles

Une file contient deux piles :

- une première nommée *entrée*, qui représente la queue de la file,
- une seconde nommée *sortie*, qui représente la tête de la file.

Un élément de la file est dans une seule des deux piles. Il suit les étapes suivantes :

1. pour l'enfiler, on l'empile au sommet de la pile *entrée*,
2. on le déplace dans la pile *sortie*,
3. pour le défiler, on dépile la pile *sortie*.

Pour enfiler, on empile sur la pile *entrée*. Donc cette dernière est dans l'ordre inverse de la file. Les éléments en tête sont les derniers de la file. Pour pouvoir défiler dans l'ordre, il faut que la pile *sortie* soit dans le même ordre que la file. Donc les premiers éléments à avoir été insérés dans la file. se situent en tête de *sortie*. Lorsque la pile *sortie* est vide, on déplace *entrée* sur *sortie*, ce qui inverse l'ordre.

Les primitives `creerFile`, `estVide` et `enfiler` sont en temps constant. En revanche, `valeur` et `dépiler` sont potentiellement linéaire puisqu'elle peuvent nécessiter un déplacement d'une pile sur l'autre. Ce qui leur permet d'être tout de même des primitives est que leur complexité amortie est constante.

La complexité amortie d'une succession d'étapes est la complexité totale des étapes ramenée sur le nombre d'étape. Grossièrement, pensez à une facture mensuelle. Vous ne payez pas tous les jours, mais à la fin du mois, vous payez d'un coup une grosse somme pour tous les jours du mois. Ici, en faisant transiter n éléments par une file, certaines étapes seront linéaires mais il y a en a peu, ce qui permet d'obtenir une complexité totale qui est aussi linéaire au lieu de quadratique. Si on divise par le nombre d'élément, la complexité amortie (par appel de primitive) est donc constante. Pour le justifier, vous pouvez remarquer que chaque élément ne demande que six appels (donc un nombre constant) à des primitives :

- `estVide` pour vérifier si il faut déplacer la piles *entrée*,
- `estVide` pour vérifier si la pile *entrée* n'est pas vide (et donc la file),
- `valeur` pour le récupérer dans *entrée* afin de le placer dans *sortie*,
- `dépiler` *entrée* pour le supprimer de cette pile,
- `empiler` pour le rajouter à *sortie*,
- `dépiler` *sortie* pour le supprimer de cette pile.

Algorithm 11: creerFile

```
1 F ← file
2 F.input = creerPile()
3 F.output = creerPile()
```

Algorithm 12: estVide

Input: F une file

```
1 return estVide(F.entree) et estVide(F.sortie)
```

Algorithm 13: valeur

Input: F une file

```
1 if estVide(F.sortie) then
2 |   if estVide(F.entree) then
3 |   |   #erreur
4 |   deplacer(F.entree, F.sortie)
5 return valeur(F.sortie)
```

Algorithm 14: depiler

Input: F une file

```
1 if estVide(F.sortie) then
2     if estVide(F.entree) then
3         #erreur
4     deplacer(F.entree, F.sortie)
5 return depiler(F.sortie)
```

Algorithm 15: empiler

Input: F une file, x un objet

```
1 empiler(F.entree, x)
2 return F
```

7 Implémentation des piles à partir de tableaux

Pour cette implémentation, on peut commencer avec un tableau de taille fixe. On a des cases réservées en mémoire (*capacity*), et à un instant donné, le nombre d'élément de la pile est *size* et on utilise les *size* premières cases du tableau (le bas de la pile étant d'indice 0).

Dans le cours, vous avez vu une version très similaire. Au lieu d'avoir le nombre d'éléments de la pile, vous aviez l'indice du sommet de la pile. On a $sommet = size - 1$, il faudra donc penser à adapter selon le choix que vous faites.

Enfin, si vous voulez rendre vos tableaux dynamiques, pensez rajouter une réallocation de la mémoire quand *size* dépasse *capacity*. Pour rappel, une réallocation peut demander de copier le tableau. On évite donc de devoir le faire à chaque étape en demandant de rajouter une seule case au tableau. Au lieu de cela, on multiplie par deux le nombre de cases. La complexité amortie est constante. L'idée est la même que dans la section 6 : la somme des opérations requises pour les copies sera linéaire.

```c
#define TAILLE_INIT 8

struct stack{
  obj * values;
  int size;
  int capacity;
};
typedef struct stack * stack;

stack creerPile(){
  stack P = (stack)malloc(sizeof(struct stack));
  if(P==NULL) error();
  P->values = (obj*)malloc(sizeof(obj)*TAILLE_INIT);
  if(P->values==NULL) error();
  P->size = 0;
  P->capacity = TAILLE_INIT;
  return P;
}
```

```
bool estVide(stack P){
  if(P==NULL) error();
  return P->size==0;
}

obj valeur(stack P){
  if(P==NULL) error();
  if(size==0) error();
  return P->values[size-1];
}

stack depiler(stack P){
  if(P==NULL) error();
  if(size==0) error();
  P->size--;
}

stack empiler(stack P, obj x){
  if(P==NULL) error();
  if(P->size==P->capacity){
    P->capacity *= 2; //on multiplie par 2 la capacite
    P->values = (obj*)realloc(P->values,sizeof(obj)*P->capacity);
    if(P->values==NULL) error();
  }
  P->size++;
  P->values[size] = x;
}
```

8 Implémentation des piles à partir de listes simplement chaînées

Ici, pas besoin de réecrire les primitives `estVide` et `valeur` puisqu'elles sont identiques.

```
typedef list stack;

stack creerPile(){
  return creerListe();
}

stack depiler(stack L){
  return supprimerDebut(L);
}

stack empiler(stack L, obj x){
  return insererDebut(L,x);
}
```

9 Implémentation des files à partir de piles

On reprend les algorithmes de la section 6. Notez que contrairement au mode utilisateur, il n'est pas nécessaire, en concepteur, de récupèrer le retour des primitives `empiler` et `depiler`. En effet, si on utilise une des deux implémentations des piles proposées ci-dessus, les structures de piles sont modifiées.

```
struct queue{
  stack input;
  stack output;
};
typedef struct queue * queue;

queue creerFile(){
  F = (queue)malloc(sizeof(struct queue));
  if(F==NULL) error();
  F->input = creerPile();
  F->output = creerPile();
  return F;
}

bool estVide(queue F){
  return estVide(F->input) && estVide(F->ouput);
}

obj valeur(queue F){
  if(estVide(F->output)){
    if(estVide(F->input)) error();
    while(not estVide(F->input)){
      v = valeur(F->input);
      depiler(F->input);
      empiler(F->ouput,v);
    }
  }
  return valeur(F->output);
}

queue defiler(queue F){
  if(estVide(F->output)){
    if(estVide(F->input)) error();
    while(not estVide(F->input)){
      v = valeur(F->input);
      depiler(F->input);
      empiler(F->ouput,v);
    }
  }
  depiler(F->output);
  return F;
}
```

```
queue enfiler(queue F, obj x){
  empiler(F->input,x);
  return F;
}
```
