TD n° 8 : Piles et files

EXERCICE 1 *Copies*

On dispose d'une pile de copies de deuxième année, triées par noms, qui proviennent de MP ou de MP*. Comment procéder pour transformer cette pile en une pile dans laquelle toutes les copies de MP sont situées au-dessus de celles de MP*, en conservant l'ordre relatif des copies, c'est-à-dire que les copies de MP et de MP* sont toujours bien triées (sans les trier à nouveau, évidemment) ? Décrire un tel algorithme et en donner la complexité.

EXERCICE 2 (*Rotations*)

1. Décrire un algorithme qui réalise la rotation d'une pile, c'est-à-dire que l'élément au sommet se retrouve tout en bas de la pile et donner sa complexité.

$$\begin{vmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{vmatrix} \longrightarrow \begin{vmatrix} a_1 \\ \vdots \\ a_n \\ a_0 \end{vmatrix}$$

   On pourra utiliser une pile auxiliaire.
2. Décrire un algorithme qui réalise la rotation inverse.
3. Faire de même avec la structure de file.

EXERCICE 3 *Mots bien parenthésés*

On considère des mots sur l'alphabet $\{(,[,),]\}$. Par exemple $(][(([($ et $)[()]$ sont deux mots. Le mot vide ne comportant aucune lettre est noté ε. On dit qu'un mot est *bien parenthésé* s'il peut être réduit au mot vide en supprimant successivement des paires adjacentes de parenthèses de même type. Par exemple le mot $([()[]])()$ est bien parenthésé car

$$([()[]])() \to ([[]])() \to ([])() \to ()() \to () \to \varepsilon$$

Décrire un algorithme qui vérifie si un mot est bien parenthésé.

EXERCICE 4 *Parcours d'abres*

On définit le type suivant pour les arbres binaires :

OCAML
```
type arbre =
  | Nil
  | Noeud of int * arbre * arbre
```

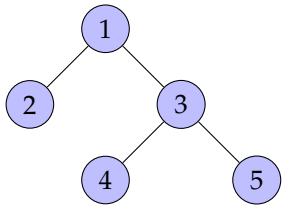


FIGURE 1 – Exemple d'arbre.

On considère le code CAML suivant :

OCAML
```
let parcours arbre =
  let rec traite pile =
    if not pile_est_vide pile then begin
      let tete = depiler pile in
      match tete with
      | Nil -> traite pile
      | Noeud (elt, fg, fd) ->
         print_int elt;
         empiler fd pile;
         empiler fg pile;
         traite pile
    end
  in
  let pile = creer_pile_vide () in
  empiler arbre pile;
  traite pile
```

1. Simuler l'exécution de cette fonction sur l'arbre de la figure 1 en représentant l'évolution de la pile.
2. Quel type de parcours est effectué par cet algorithme ?
3. Pourquoi le fils droit est-il empilé avant le fils gauche ?
4. Réécrire cette fonction à l'aide d'une boucle conditionnelle **while** plutôt qu'une fonction auxiliaire récursive.
5. On remplace la structure de pile par une structure de file. Simuler l'exécution de ce nouvel algorithme. Que faut-il modifier d'autre pour obtenir un parcours en largeur ?

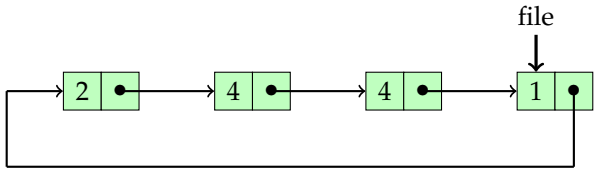
On considère maintenant des arbres n-aires dont le type en Caml est

OCaml
```
type arbre = {etiquette : int; fils : arbre list};;
```

6. Adapter les fonctions précédentes pour écrire un parcours en profondeur préfixe et un parcours en largeur pour les arbres n-aires.

EXERCICE 5 *Implémentation d'une file circulaire*

En cours nous avons étudié l'implémentation d'une file à l'aide d'un tableau et de deux indices. Un inconvénient de cette implémentation est que la taille de la file est alors bornée. Dans cet exercice on se propose d'utiliser une liste chaînée circulaire, c'est-à-dire que le dernier élément de la liste pointe sur le premier. Pour pouvoir insérer et extraire en temps constant, il suffit de conserver un pointeur sur le dernier élément de la liste.



On propose d'implémenter le type `'a file` de la manière suivante :

OCaml
```
type 'a cellule = {
  element : 'a;
  mutable suivante : 'a cellule
}
type 'a file =
  | Vide
  | Dernier of 'a cellule

exception FileVide
```

1. Implémenter la fonction `creer_file_vide : 'a file` ainsi que la fonction `file_est_vide : 'a file -> bool`.

Pour enfiler un élément il y a une difficulté lorsque la file est vide. En effet, il faut créer une cellule qui pointe... vers elle-même ! C'est bien évidemment récursif et il y a donc une solution toute élégante en CAML. Par exemple pour créer une cellule contenant un élément `e` qui pointe vers elle-même on peut écrire :

OCaml
```
let rec c = {element = e; suivante = c} in c
```

2. Implémenter la fonction `enfiler : 'a -> 'a file -> 'a file`.
3. Implémenter la fonction `defiler : 'a file -> ('a * 'a file)`.
4. Quelle est la complexité des différentes opérations ?