Feuille de TD N^o 2 : Listes-Piles

1 Listes-Piles : listes chaînées

Les opérations élémentaires sont :

- Tester si une liste est vide : Test `l == NULL`. À la rigueur, utilisation de la fonction `bool Estvide(liste l)`. Des écritures telles que `l == Vide`, `l est Vide`, `l == ∅`, `l == []` sont tolérées (pseudocode)
- Initialisation d'une liste à vide : Affectation `l = NULL ;` À la rigueur : `void Initvide(inout liste *L)`. Des écritures telles que `l = Vide ;`, `l = ∅ ;`, `l = [] ;` sont tolérées.
- Premier élément d'une liste : `(*l).valeur` ou `l->valeur`. À la rigueur : `élément premier(liste l)` La tentative d'accéder au premier élément d'une liste vide entraîne un plantage (Segmentation Fault).
- Ajout d'un élément en tête de liste. Pour cela, il y a une fonction et une procédure :
  • La fonction `liste ajoute(x,liste l)` rend un pointeur liste pointant vers un nouveau bloc contenant la valeur x et un champs suite copie de l. La liste l n'est pas modifiée, n'est pas recopiée, son espace mémoire est partagé avec celui du résultat de `ajoute`. Cf. code dans le cours dans lequel il y a un (et un seul) malloc.
  • La procédure `empile(x,inout *L)` modifie la liste en insérant x en tête. Elle équivaut à `l = ajoute(x,l) ;`
- Liste privée de son premier élément :
  • Accès à la suite de la liste : `(*l).suite` ou `l->suite`. À la rigueur : `liste suite(liste l)`
  La liste `l` n'est pas modifiée, elle n'est pas recopiée, son espace mémoire est partagé avec celui du résultat de `suite`. Il n'y a pas de désallocation mémoire (pas de free). C'est comme si on tournait la page d'un livre.
  • La procédure `depile(inout *L)` modifie la liste en enlevant son premier élément, lequel est désalloué, i.e. rendu à la mémoire. Cf. code dans le cours, qui inclut un free. C'est comme si on arrachait la page d'un livre.
  • "PointeurSuite" : Lorsque l'on appelle récursivement sur suite(l) une procédure qui modifie les listes, on ne peut pas utiliser le passage par valeur. Il faudra donc faire un "PointeurSuite" et montrer les * et les &. Il faudra utiliser et comprendre des écritures telles que &(**L).suite ou &(*L)->suite.

  La tentative de faire suite, depile ou PointeurSuite sur une liste vide entraîne un plantage (Segmentation Fault).

Écrire les fonctions et procédures suivantes :

1. **Doubleton (l)** qui rend vrai ssi l contient exactement deux éléments (éventuellement identiques)
2. **Affiche (l)** qui affiche les éléments dans l'ordre.
3. **ehciffA (l)** qui affiche les éléments dans l'ordre inverse (dit aussi "ordre miroir")
4. **Copie (l)** qui rend une liste représentant la même liste mathématique que l, mais avec des blocs indépendants et neufs.
5. **LongueurK (k,l)** qui rend vrai ssi la longueur de l est k
6. **ComptageOccurrences** qui prend deux listes m et t de lettres, m étant interprété comme un mot, et t comme un texte, et rend le nombre d'apparitions du mot m dans le texte t. Exemple:
   $ComptageOccurrences([a,b,c,a,b],[b,a,b,c,a,b,b,b,a,a,b,c,a,b,c,a,b])$ rend 3.
7. **AjouteTrie(…)** qui prend pour arguments x et une liste l supposée triée et y insère x (le résultat étant trié, bien sûr). Faire deux versions: une version fonction (rend la nouvelle liste) et une version procédure (modifie l).
8. **ElimineEnPremiereOccurrence** qui prend en argument une liste l, un élément x et modifie l en y éliminant la première occurrence de x. (Si x n'apparaît pas dans l, la procédure ne fait rien.)
9. **ElimineToutesOccurrences** qui prend en argument une liste l, un élément x et modifie l en y éliminant toutes les occurrence de x.
10. **ElimineDerniereOccurrence** qui prend en argument une liste l, un élément x et modifie l en y éliminant la dernière occurence de x. (Si x n'apparaît pas dans l, la procédure ne fait rien.) Ne faire qu'une seule passe.
11. **DernierePosition(l,x)** rend la position de x dans la liste L (par exemple 4 si $x = 8$ et $l = [4,3,1,8,9]$). Si x n'apparaît pas dans la liste, la fonction rendra 0. Si x apparaît plusieurs fois, la fonction rendra la position de la dernière occurence. Écrire une version récursive sans sous-fonction, une itérative et une récursive terminale.
12. **Pif(l)** qui rend le nombre d'éléments qui sont égaux à ((leur position depuis le début) fois (leur position depuis la fin)). Exemple, Pif([2,8,7,0,5]) rendra 2 (cf. le 8 (=2*4) et le 5 (=5*1)). Ne faire qu'une seule passe.
13. **Swap (…)**, qui intervertit les deux premiers blocs de l. Si l vaut [23,47,19,42,69,97,33] avant l'appel , elle vaut [47,23,19,42,69,97,33] après.

14. Que calcule la fonction Truc ? :

```
Pof (inout * P, in l, out * Resultat)
si L est vide alors *Resultat = vrai
else   Pof(P, l->suite, Resultat)
       si *P->valeur ≠ l->valeur alors *Resultat = faux
       *P = (*P)->suite
```

```
bool Truc(l)
bool resultat ;
liste p = Ll ;
Pof (inout & p, in l, out & resultat) ;
rendre resultat ;
```

15. **fusion (l1,l2)** qui rend l'union triée des listes $l1$, $l2$ en argument, supposées triées.

16. **Fmiroir** et **Pmiroir** qui prennent une liste en argument. La première est une fonction qui rend une copie miroir. La seconde est une procédure qui transforme l en son miroir. Miroir de *algorithme*, c'est *emhtirogla*.

17. Une liste $l = (x_1, ..., x_n)$ est un interclassement de l_1 et de l_2 ssi on peut partitionner $[1..n]$ en deux parties I et J telle que l_1 soit la sous-liste des éléments de L dont les indices sont dans I, l_2 celle des éléments dont les indices sont dans J. Par exemple, la liste des interclassements de $[1, 2, 3]$ et de $[4, 5]$ est

    $[[1, 2, 3, 4, 5], [1, 2, 4, 5, 3], [1, 4, 5, 2, 3], [4, 5, 1, 2, 3], [1, 2, 4, 3, 5], [1, 4, 2, 5, 3], [4, 1, 5, 2, 3], [1, 4, 2, 3, 5], [4, 1, 2, 5, 3], [4, 1, 2, 3, 5]]$

    Écrire le pseudo-code de **Interclassements** qui prend en argument l_1 et l_2 et rend la liste des interclassements de l_1 et de l_2 (Si l_1 et l_2 ont des éléments en commun, il est autorisé et normal d'obtenir plusieurs fois une même liste. Par exemple, $[1, 2, 2, 3]$ devrait apparaître deux fois dans `Interclassements` de $[1, 2]$ et de $[2, 3]$)

Feuille de TD N^o 3 : Listes chaînées et pointeurs

2 & et *

Comparez $*(\&n)$ et n. Comparez $\&(*p)$ et p.

3 Echange

Que font les programmes suivants ?

```
void echange_1 (int x, int y)
{
int tmp ;
    tmp = x ;
    x = y ;
    y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_1(a,b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_2 (int *x, int *y)
{
int *tmp ;
     tmp = x ;
     x = y ;
     y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_2(&a,&b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_2 (int *x, int *y)
{
int *tmp ;
     tmp = x ;
     x = y ;
     y = tmp ;
 }

void main()
{
int a = 2 ;
int b = 3 ;
echange_2(a,b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_3 (int *x, int *y)
{
int tmp ;
    tmp = *x ;
    *x = *y ;
    *y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_3(&a,&b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_3 (int *x, int *y)
{
int tmp ;
     tmp = *x ;
     *x = *y ;
     *y = tmp ;
}

void main()
{
int a = 2 ;
int b = 3 ;
echange_3(a,b) ;
printf("%d %d\n",a,b) ;
}
```

```
void echange_4 (int *x, int *y)
{
int *tmp ;
     *tmp = *x ;
     *x = *y ;
     *y = *tmp ;
 }

void main()
{
int a = 2 ;
int b = 3 ;
echange_4(&a,&b) ;
printf("%d %d\n",a,b) ;
}
```

4

Que pensez-vous des codes suivants ?

```
void FactBis (int n, out int *R)
si n == 0 alors *R = 1 ;
sinon { *R = (*R)*n ;
        FactBis(n-1,R) ; }
```

```
int Factorielle1 (int n)
int r ; int * p = & r ;
FactBis(n,p) ;
rendre r ;
```

```
int Factorielle2 (int n)
int * p = (int*) malloc(sizeof(int)) ;
FactBis(n,p) ;
rendre *p ;
```