TD de Système n° 2

I) Tableaux bidimensionnels

Exercice 1 : allocation d'un tableau à deux dimensions

Proposer une représentation d'un tableau à deux dimensions qui stocke les entrées de façon contiguë en mémoire. Comment peut-on ensuite parcourir ce tableau sans utiliser la notation [ ]? Donner l'exemple du remplissage du tableau par des0 , puis celui de l'affichage d'une ligne, et enfin de l'affichage d'une colonne.

Exercice 2 :

En utilisant la resprésentation de l'exercice1, écrire une fonction qui permute les lignes d'un tableau bidimensionel selon une permutation passée en paramètre (dont on suppose qu'elle a la bonne taille). La permutation est représentée, comme dans le TP1, par une structure :

```
#define n AX 50

typedef struct {
  int taille;
  int tab AX];
} permutation;
```

Peut-on déclarer un tableau bidimensionnel classique et le passer en paramètre à cette fonction ?

II) Listes chaînées

Le but de cette série d'exercices est de manipuler des listes (simplement) chaînées, contenant des entiers. Une liste chaînée sera représentée par un pointeur vers une structure de la forme :

```
struct cell {
  int val;
  struct cell *suiv;
};
```

La liste vide sera représentée par un pointeur vers NULL. Toute création de cellule nécessite un malloc , et toute destruction de cellule un free. La plupart de ces fonctions peuvent être définies par récurrence, et il est conseillé de trouver une écriture de cette forme à chaque fois qu'elle est envisageable.

Exercice 3 :

Un exemple de code utilisant chacune des fonctions ci-dessous est donné en fin d'exercice.

1. Ecrire une fonction :

```
int longueur(struct cell *p
```

Renvoyant le nombre de cellules de la liste `p` . Cette fonction peut être définie par récurrence.

2. Ecrire une fonction :

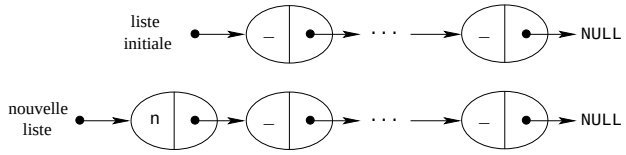
   `void affiche_liste(struct cell *p)`

   Cette fonction devra afficher la suite des clefs de chaque cellule de la liste `p` , ou bien le message `Liste vide` si la liste est vide. Cette fonction peut être définie par récurrence.

3. Ecrire une fonction :

   `struct cell *ajoute_debut(struct cell *p, int n)`

   ajoutant au début de la liste chaînée `p` une nouvelle cellule de clef `n`, et renvoyant l'adresse de la première cellule de la nouvelle liste.

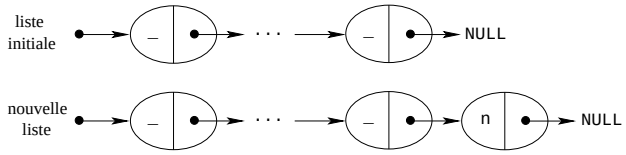


   Noter que l'adresse renvoyée est toujours celle de la cellule créée.

4. Ecrire une fonction :

   `struct cell *ajout_fin(struct cell *p, int n)`

   ajoutant à la fin de la liste chaînée `p` une nouvelle cellule de clef `n`, et renvoyant l'adresse de la première cellule de la nouvelle liste. Cette fonction peut être définie par récurrence.



   Noter que l'adresse est la valeur initiale de `p` si la liste initiale est non vide, l'adresse de la cellule créée si la liste initiale est vide.

5. Ecrire une fonction :

   `struct cell *supprime_debut(struct cell *p)`

   Cette fonction doit réaliser l'inverse de `ajoute_debut`. Elle doit supprimer la première cellule de liste `p` - en libérant l'espace-mémoire reservé par cette cellule - et renvoyer l'adresse de la première cellule de la nouvelle liste. Elle ne devra rien faire si la liste est vide, et renvoyer dans ce cas la même liste, c'est-à-dire un pointeur nul.

6. Ecrire une fonction :

   `struct cell *suppr_fin(struct cell *p)`

   Cette fonction doit réaliser l'inverse de `ajout_fin`. Elle doit supprimer la dernière cellule de liste `p` - en libérant l'espace-mémoire alloué à cette cellule - et renvoyer l'adresse de la première cellule de la nouvelle liste.

   La fonction devra ne rien faire si la liste est vide, et renvoyer dans ce cas la même liste, c'est-à-dire un pointeur nul. Elle peut être définie par récurrence.

Exemples d'appels

```
stru t *    p    NULL;
a i n*r(p                    /  a i n* *Li st* v i*d*  /

p   *p i *r(p ,1 ;
p   *p i *r(p ,0 ;
pr i nt (* *n*, ong u*ur(p     /  a i n* *2*           /
a i n*r(p  ;                  /  a i n* *0 1*         /

p   a_ put*r(p ,2 ;
a i n*r(p  ;                  /  a i n* *0 1 2*       /

p   *p i *r(p  ;
a i n*r(p  ;                  /  a i n* *1 2*         /

p   suppr i *r(p  ;
a i n*r(p  ;                  /  a i n* *1*           /
```

Exercice 4 :

1. Ecrire une fonction

   ```
   vo i**tru i r*(stru t *    p
   ```

   détruisant, en libérant l'espace mémoire qui leur est alloué, chacune des cellules de la liste `p` . Donner deux versions de cette fonction :
   – une version écrite avec **p i *r,
   – une version sans autre fonction externe que r**, récursive.

2. Ecrire une fonction

   ```
   stru t *     op i *r(stru t *    p
   ```

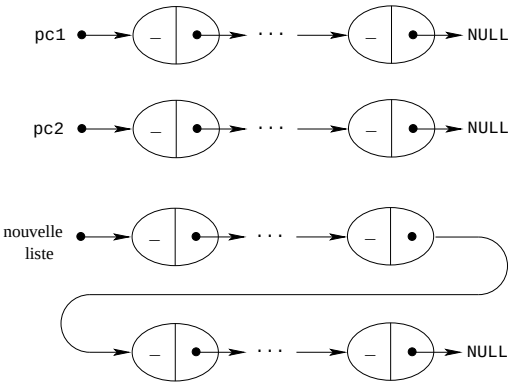
   créant une copie conforme de la liste `p` , et renvoyant l'adresse de la première cellule de cette copie.

   Cette fonction est beaucoup plus simple à définir par récurrence, avec *p i *r comme seule fonction externe. On évitera de recourir à une itération de la fonction `a_put*r`, la méthode étant trop coûteuse en temps.

3. Ecrire une fonction

   ```
   stru t *     ma i n*r(stru t *    p 1, stru t *    p 2
   ```

   chaînant la liste `p 1` avec la liste `p 2`, et renvoyant l'adresse de la première cellule de la liste résultante. Cette fonction ne devra faire aucune allocation. Elle peut être définie par récurrence. Attention au cas où la première liste est vide.


pc1
NULL
pc2
NULL
nouvelle
liste
NULL