TD de Système n° 2

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 clef;
  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 1 :

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

1. Ecrire une fonction :

   `int longueur(struct cell *pc)`

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

2. Ecrire une fonction :

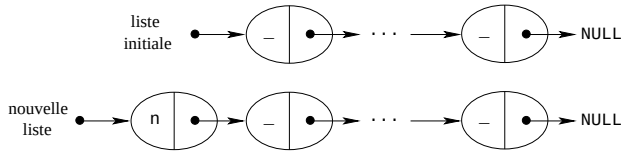
   `void afficher(struct cell *pc)`

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

3. Ecrire une fonction :

   `struct cell *empiler(struct cell *pc, int n)`

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

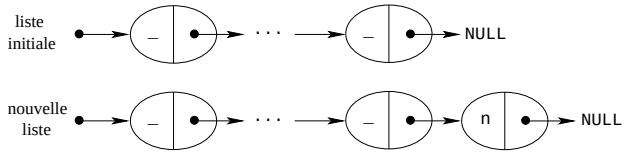


   Noter que l'adresse renvoyée est toujours celle de la cellule créée.

4. Ecrire une fonction :

   `struct cell *ajouter(struct cell *pc, int n)`

   ajoutant à la fin de la liste chaînée `pc` 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 `pc` 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 *depiler(struct cell *pc)`

   Cette fonction doit réaliser l'inverse de `empiler`. Elle doit supprimer la première cellule de liste `pc` - 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 *supprimer(struct cell *pc)`

   Cette fonction doit réaliser l'inverse de `ajouter`. Elle doit supprimer la dernière cellule de liste `pc` - 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

```
struct cell *pc = NULL;
afficher(pc)                 /* affiche "Liste vide" */

pc = empiler(pc,1);
pc = empiler(pc,0);
printf("%d\n",longueur(pc))  /* affiche "2"          */
afficher(pc);                /* affiche "0 1"        */

pc = ajouter(pc,2);
afficher(pc);                /* affiche "0 1 2"      */

pc = depiler(pc);
afficher(pc);                /* affiche "1 2"        */

pc = supprimer(pc);
afficher(pc);                /* affiche "1"          */
```

Exercice 2 :

1. Ecrire une fonction

   `void detruire(struct cell *pc)`

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

2. Ecrire une fonction

   `struct cell *copier(struct cell *pc)`

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

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

3. Ecrire une fonction

   `struct cell *chainer(struct node* pc1, struct node* pc2)`

   chaînant la liste `pc1` avec la liste `pc2`, 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.