Programmation en C

Juin 2006 - 2nde session

Bien qu'inutiles, les notes de cours sont autorisées, pas les livres. N'oubliez pas de commenter votre code.

Exercice I

Une matrice carrée est un *étalement* si, pour chaque éléments e de sa diagonale (i) tous les éléments situés à droite de e et sur la même ligne que e sont égaux à e, et (ii) tous les éléments situés sous e et sur la même colonne que e sont égaux à e. Par exemple, la matrice suivante est un étalement :

$$\begin{pmatrix} 2 & 2 & 2 & 2 \\ 2 & 42 & 42 & 42 \\ 2 & 42 & 17 & 17 \\ 2 & 42 & 17 & 5 \end{pmatrix}$$

Ecrire une fonction `int etalement(int m[TAILLE][TAILLE])` renvoyant `1` si `m` est un étalement, et `0` sinon. On supposera que `TAILLE` est une constante quelconque déclarée par un `#define` en début de programme.

Exercice II

Un texte écrit peut contenir plusieurs sortes de parenthèses, des parenthèses rondes, des accolades, des crochets : `()`, `{}`, `[]`. Ces parenthèses peuvent être imbriquées, comme par exemple dans `"f(tab[(i + j)/2])"`.

Test de bon parenthésage

Un texte est *bien parenthésé* s'il est possible de cocher progressivement deux par deux toutes ses parenthèses, en respectant la contrainte suivante :

- à chaque étape, on ne peut cocher un couple de parenthèses que si :
  - la plus à gauche est ouvrante, non encore cochée,
  - la plus à droite est fermante, non encore cochée,
  - les deux sont de même sorte : `(` et `)`, ou bien `{` et `}`, ou bien `[` et `]`,
  - entre les deux ne se trouve aucune parenthèse non cochée.

Par exemple, `(a[b(c)d]{e})` est bien parenthésé :

$$\begin{array}{ll}
 & \texttt{(a[b(c)d]\{e\})} \\
\rightarrow & \texttt{(a[b }\cancel{\texttt{(}}\texttt{c}\cancel{\texttt{)}}\texttt{d]\{e\})} \\
\rightarrow & \texttt{(a[b}\cancel{\texttt{(}}\texttt{c}\cancel{\texttt{)}}\texttt{d]}\cancel{\texttt{\{}}\texttt{e}\cancel{\texttt{\}}}\texttt{)} \\
\rightarrow & \texttt{(a}\cancel{\texttt{[}}\texttt{b}\cancel{\texttt{(}}\texttt{c}\cancel{\texttt{)}}\texttt{d}\cancel{\texttt{]}}\cancel{\texttt{\{}}\texttt{e}\cancel{\texttt{\}}}\texttt{)} \\
\rightarrow & \cancel{\texttt{(}}\texttt{a}\cancel{\texttt{[}}\texttt{b}\cancel{\texttt{(}}\texttt{c}\cancel{\texttt{)}}\texttt{d}\cancel{\texttt{]}}\cancel{\texttt{\{}}\texttt{e}\cancel{\texttt{\}}}\cancel{\texttt{)}}
\end{array}$$

Par contre, $\texttt{[a(b)} \rightarrow \texttt{[a}\cancel{\texttt{(}}\texttt{b}\cancel{\texttt{)}}$ ou $\texttt{a(b)\}} \rightarrow \texttt{a}\cancel{\texttt{(}}\texttt{b}\cancel{\texttt{)}}\texttt{\}}$ ou encore `a[b{c]}` sont mal parenthésés. Dans chaque cas, il est impossible de cocher toutes les parenthèses du texte en respectant la contrainte ci-dessus.

Implémentation

Prenez le temps d'analyser la méthode ci-dessus, et de refêchir à la manière dont elle peut être implémentée. Discutez brièvement vos choix d'implémentation.

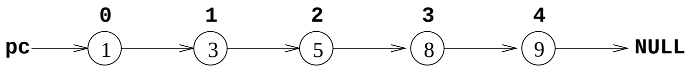
Ecrire ensuite la fonction `int bien_par(char *s)`, renvoyant `1` si la chaîne `s` est bien parenthésée et `0` sinon. Cette fonction doit préserver le contenu de la chaîne. Vous pouvez vous servir de `malloc` et `free`, de `strlen`, écrire des fonctions auxiliaires, définir des constantes, utiliser ou non la récurrence...

Exercice III

On considère le type usuel permettant de représenter les listes chaînées d'entiers :

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

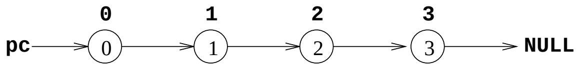
Comme d'habitude, une liste sera représentée par un pointeur vers sa première cellule, et la liste vide par un pointeur nul. Les cellules d'une liste seront numérotées à partir de `0`. Par exemple, voici la représentation de la liste contenant successivement `1`, `3`, `5`, `8` et `9`, avec ses numéros de cellules :



- Ecrire `int croissante(struct cell *pc)`, renvoyant `1` si la suite des éléments de `pc` est croissante, et `0` sinon. Par exemple, la liste ci-dessus est croissante. Par convention, une liste vide ou réduite à un élément est croissante.

  Cette fonction est plus facile à écrire par récurrence.

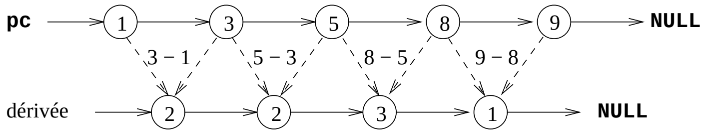
- Une liste sera dite *canonique* si la clef de chaque cellule est égale à son numéro. Par exemple, la liste ci-dessous est canonique :



  Ecrire `struct cell *canonique(int n)`. Cette fonction doit construire en mémoire une liste canonique à `n` éléments, et renvoyer un pointeur vers sa première cellule (ou `NULL` si `n` vaut `0`).

- Ecrire *par récurrence* : `struct cell *derivee(struct cell *pc)`

  Si `pc` est vide ou ne contient qu'un seul élément, cette fonction doit renvoyer `NULL`. Sinon, elle doit construire une nouvelle liste contenant successivement, pour chaque clef de `pc` à partir de la clef $n.^\circ 1$, la différence entre cette clef et la précédente.

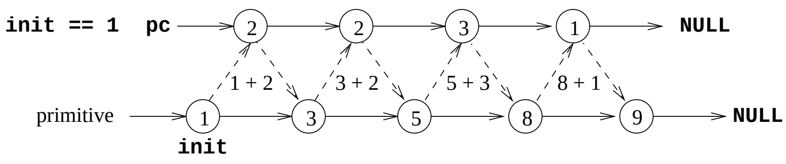


Elle doit renvoyer l'adresse de la première cellule de la liste construite.

- Ecrire *par récurrence* :

  `struct cell *primitive(int init, struct cell *pc)`

  Cette fonction est en quelque sorte l'inverse de la précédente. Elle doit construire une nouvelle liste dont la clef $n.°0$ vaut `init`. Pour $i > 0$, sa clef $n.°i$ vaut sa clef $n.°i - 1$ plus la clef $n.°i$ de `pc`.



  Elle doit renvoyer l'adresse de la première cellule de la liste construite.

- Ecrire la fonction :

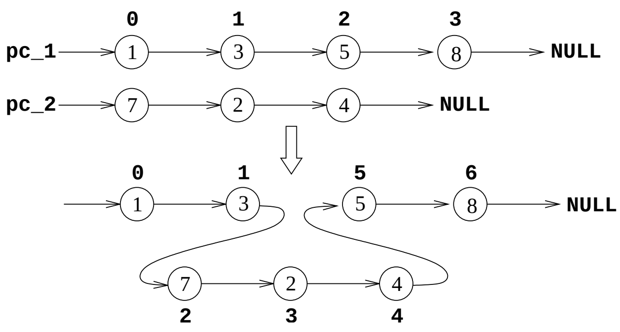
  `struct cell *inserer(struct cell *pc_1, int num, struct_cell *pc_2)`

  Sans créer de nouvelles cellules (sans aucun `malloc`), cette fonction doit modifier le chaînage des cellules de `pc_1` et `pc_2` de manière à former une chaîne de cellules contenant successivement :

  - les cellules numérotées de `0` à `num - 1` dans `pc_1` (s'il en existe),
  - les cellules de `pc_2` (s'il en existe),
  - les cellules numérotées à partir de `num` dans `pc_1` (s'il en existe).

  Elle doit renvoyer l'adresse de la première cellule de la liste résultante (ou `NULL` si les deux listes sont vides).

  Voici un exemple d'insertion avec `num` égal à `2` :



  Cette fonction peut être écrite par récurrence. Attention aux cas limites.