TP6 : manipulation de fichiers, piles et files

Algorithmique et projet de programmation — M1 mathématiques

8 novembre 2016

1 Manipulation de fichiers

On va s'atteler à recoder (une version simple de) la commande `sort` des sytèmes UNIX. C'est un petit programme qui prend en paramètre un chemin d'accès à un fichier texte et affiche à l'écran l'ensemble de ses lignes triées alphabétiquement. Ainsi, si un fichier `phonebook` contient :

```
Zaid 0673894765
Bernard 0663896735
Lea 0612673546
Clement 0678900034
Adrien 0694367833
Claudia 0642356732
```

L'exécution de la commande `sort phonebook` affichera :

```
Adrien 0694367833
Bernard 0663896735
Claudia 0642356732
Clement 0678900034
Lea 0612673546
Zaid 0673894765
```

Exercice 1. Écrire une fonction qui compare deux chaînes de caractères dans l'ordre lexicographique :

```
char compare_str (char* s, char* t);
```

Elle renverra `1` si `s` est inférieur à `t` au sens large et `0` sinon. On rappelle que `s` $\leq$ `t` si et seulement si

- `s` est un préfixe de `t`
- il existe $i \geq 0$ tel que `s[k] = t[k]` pour tout $0 \leq k \leq i-1$ et `s[i]` $\leq$ `t[i]`.

Exercice 2. Écrire une fonction adaptée des fonctions de tri vues la dernière fois :

```
void sort_str (char** strs);
```

qui trie alphabétiquement un tableau de chaînes de caractères.

Rappelons que pour traiter les fichiers textes, le langage `C` dispose des fonction suivantes, disponibles dans la bibliothèque `stdio.h` :

```c
/*
  fopen takes the file path as a string and the opening mode that
  could be multiple thing: for our use, we just consider mode to be
  "r", opening the file in read-only mode.

  It returns a pointeur to a FILE structure, that we shall consider
  opaque.

  E.g.: FILE* f = fopen("path/to/my/file", "r");
*/
FILE* fopen (const char* path, const char* mode);

/*
  fgetc takes a pointer to a FILE and returns either a character just
  read in the file f or EOF, a constant meaning the End Of the File is
  reached.

  Beware that, when it does returns a character, it casts it to an
  int.

  E.g. : if( (c = fgetc(f)) != EOF ) printf("%c", (char) c);
*/
int fgetc (FILE* f);

/*
  fgets takes a string s, a buffer length size and a pointer to a FILE
  f; it reads in f a string of length size-1 (stopping to the first
  '\n' or EOF if encountered) and puts it inside s. It writes a '\0'
  as the last character of s. It returns s if everything went right
  and NULL if not or if EOF is encountered before reading anything.

  E.g. : char s[10]; if ( fgets(s,10,f) != NULL ) printf("%s",s);
*/
char* fgets (char* s, int size, FILE* f);

/*
  rewind puts the current position in the file back to the beginning
  of the file pointed by f.

  It behaves like closing (see below) and reopening the file.
*/
void rewind (FILE* f);

/*
  fclose frees the memory allocated for the FILE structure pointed by
  f; it returns 0 upon success (and something else upon failure we are
  not interested in here).
*/
int fclose(FILE* f);
```

Exercice 3. En déduire un programme implémentant la commande `sort` décrite précédemment. En particulier, on déterminera le nombre de lignes et la taille maximale des lignes du fichier passé en paramètre pour allouer un tableau de taille suffisante.

2 Piles et files d'attente

La structure de pile (*stack* en anglais) est très utile en informatique. Notamment, elle modélise bien la forme de la mémoire et est donc particulièrement adaptée à l'écriture d'interpréteur (cf. section 3). Rappelons qu'une pile est une structure de données modélisant un ensemble fini et munie de deux fonctions dédiées : `stack_push` qui va ajouter un élément à la pile (on dit parfois *empiler*), et `stack_pop` qui supprime et retourne le dernier élément ajouté à la pile (on dit parfois *dépiler*).

Il existe de nombreuses manières d'implémenter les piles. Néanmoins, la plupart du temps, on utilise une liste chaînée : `stack_push` ajoute l'élément en tête de liste, et `stack_pop` retire la tête de liste. Ces deux actions se font donc en temps constant[1].

Exercice 4. Écrire une structure modélisant les piles de chaînes de caractères. (On peut évidemment retourner chercher du code des séances précédentes...)

Exercice 5. Écrire les fonctions d'empilement et dépilement :

```
stack_t stack_push (char *string, stack_t s);
char *stack_pop (stack_t s); // here 's' gets modified
```

Les files d'attente (*queue* en anglais) jouent un rôle dual à celui des piles : elles modélisent tout autant des ensembles finis, et viennent également avec deux fonctions dédiées `queue_push` et `queue_pop`. Mais si `queue_push` ajoute bien un élément à la file d'attente, la fonction `queue_pop` supprime et retourne l'élément le plus anciennement ajouté à la file. C'est-à-dire que si on ajoute à une file vide, d'abord 1 puis 2 puis 3, la fonction `queue_pop` doit retourner 1 (alors que la même opération sur une pile aurait retourné 3).

La structure classique implémentant les files d'attente sont les listes *doublement* chaînées. Cette structure est très proche des listes chaînées mais chaque cellule comporte deux pointeurs : un vers la cellule suivante (comme pour les listes chaînées) et un vers la cellule précédente. Remarquons alors qu'il n'y a plus de *sens* dans une liste doublement chaînée (qui est la tête ?). Ainsi, au même titre qu'on identifiait une liste chaînée avec un pointeur sur la première cellule, on identifie une liste doublement chaînée avec un couple de pointeurs, un pour chaque extrêmité de la liste. La fonction `queue_push` ajoute alors un élément à une extrêmité, tandis que `queue_pop` retire un élément à l'autre.

Exercice 6. Écrire une structure modélisant les files d'attente de chaînes de caractères. Indice : cela aura la forme suivante

```
typedef struct cell_s cell_t;
typedef struct queue_s queue_t;

struct cell_s {
  /* ... inspired from cells of linked list ... */
};
struct queue_s {
  cell_t* first;
  cell_t* last;
};
```

1. Et c'est ça le plus important : quand on manipule des piles, on est amener à empiler et dépiler un très grand nombre de fois.

Exercice 7. Écrire les fonctions :

```
queue_t queue_push (char *string, queue_t q);
char *queue_pop (queue_t q); // here 'q' gets modified
```

3 Pour le 15 novembre 2016

À partir de maintenant, le projet doit être la priorité. Aussi les exercices à rendre d'une séance sur l'autre deviennent-t-ils facultatifs. Néanmoins, j'invite quiconque ayant le temps de les faire et de les envoyer par mail comme auparavant.

Exercice 8 (Calculatrice)**.** Le but de cet exercice est d'implémenter une calculatrice en utilisant la syntaxe dite *polonaise*. Dans cette syntaxe, on marque les opérations $+, *, -, /$ en préfixe de ces arguments : par exemple, on écrit `+ 1 2` pour $1 + 2$. Cela a l'avantage de ne pas avoir à parenthèser son calcul. En effet, en notation habituelle (on dit notation *infixe*), le calcul $1 + 2 * 3$ peut-être interprété comme : $(1 + 2) * 3$ ou $1 + (2 * 3)$; la notation polonaise les distingue d'emblée : le premier étant `* + 1 2 3` et le deuxième `+ 1 * 2 3`.

1. Écrire une fonction

   ```
   stack_t str_to_stack (char *s);
   ```

   qui prend en paramètre une chaîne de caractères contenant une suite d'opérations en notation polonaise et qui retourne une pile de chaînes de caractères contenant l'empilement successif des chaînes de caractères rencontrées (en considérant qu'un espace sépare deux telles chaîens). Par exemple, dépiler et afficher tous les éléments de `str_to_stack("+ 1 * 2 3")` doit donner `3 2 * 1 +`.

2. Écrire une fonction récursive

   ```
   int compute (stack_t s);
   ```

   qui maintient une deuxième pile `t` de la manière suivante tant que `s` n'est pas vide :

   - on dépile `s`,
   - soit l'élément dépilé est une chaîne de caractères contenant un entier, auquel cas on l'empile sur `t`,
   - soit l'élément dépilé est un des 4 caractères suivants : `"+"`,`"*"`,`"-"`,`"/"` ; on dépile alors deux fois `t` et on effectue l'opération avec ces deux éléments dépilés, avant d'ampiler le résultat (sous forme de chaîne de caractères) sur `t`.

   Quand la pile `s` est vide, `t` doit contenir une unique case contenant le résultat (sous forme de chaîne de caractères).

3. Coder une calculatrice complète, demandant à l'utilisateur de rentrer un calcul en notation polonaise et retournant le résultat de ce calcul.

Exercice 9 (Plus difficile)**.** Dans l'exercice précédent, la fonction `compute` est obligée de manipuler des châines de caratères alors qu'on sait bien qu'on manipule en fait des entiers stockés dans de telles chaînes et des opérations. On peut éviter cela, à condition que les cellules de notre pile soit un peu plus élaborées.

En utilisant le mot clé `union`, créer un type qui peut contenir alternativement des entiers et des caractères. Implémenter les piles pour ce nouveau type. Modifier le programme précédent pour qu'il manipule des piles de ce type.