TD4 – Manipulation de pointeurs et piles

Langage C (LC4)

semaine du 22 février

1 Échange de tableaux.

Écrire une fonction qui échange deux tableaux d'entier **t** et **r**. Écrire l'appel de cette fonction dans la fonction main.

2 Concaténations

- Écrire une fonction qui prend en arguments deux tableaux et leurs tailles respectives et qui renvoie leur concaténation.
- Écrire une fonction qui prend en arguments deux chaînes de caractères et qui renvoie leur concaténation. Vous pourrez utiliser le fait qu'une chaîne de caractères se termine par $'\backslash 0'$.

3 Piles

Une pile est une structure de données dans laquelle les derniers éléments ajoutés sont les premiers à être récupérés. Une méthode pour implémenter une pile d'entiers consiste à stocker tous les éléments de la pile dans un tableau (le sommet de la pile se trouve dans la dernière case remplie de ce tableau) :

- lorsqu'on veut ajouter (empiler) un élément, on recopie tous les éléments (et le nouvel élément à la fin) dans un tableau plus grand d'une case ;
- lorsqu'on veut enlever le dernier élément (dépiler), on recopie tous les éléments sauf le dernier dans un tableau plus petit d'une case.

Cette implémentation est simple mais inefficace puisque les éléments de la pile sont recopiés à chaque opération. Il est plus astucieux de garder un tableau partiellement rempli (méthode de la pile amortie) :

- lorsque la pile déborde, plutôt que d'allouer un tableau plus grand d'une case, on alloue un tableau deux fois plus grand ;
- lorsque l'on dépile un élément, on ne recopie dans un tableau plus petit que si le tableau est aux trois quarts vide, auquel cas on divise sa taille par deux.

On utilise donc la structure suivante pour mettre en œuvre la méthode de la pile amortie :

```
typedef struct {
  int occupation;
  int capacite;
  int *elements;
} pile_amortie;
```

où `capacite` représente le nombre de cases du tableau `elements`, tandis que `occupation` représente le nombre de cases effectivement remplies. À partir de la case numéro `occupation`, il y a des cases non utilisées.

Question 1. Écrire une fonction `pile_amortie *alloue_pile_amortie()` qui crée une pile amortie de capacité initiale 0.

Question 2. Écrire une fonction `void libere_pile_amortie(pile_amortie *pile)` qui libère la mémoire occupée par la pile `pile`. On veillera à bien libérer toute la mémoire occupée.

Question 3. Écrire une fonction `void empile_pile_amortie(pile_amortie *pile, int n)` qui empile l'entier `n` sur la pile `pile`.

Question 4. Écrire une fonction `int depile_pile_amortie(pile_amortie *pile)` qui dépile la pile `pile` et renvoie l'élément dépilé.
On utilise maintenant la structure suivante pour représenter une pile, utilisant la structure de données de liste chaînée :

```
typedef struct int_elem {
  int element;
  struct pile *precedent;
} *int_pile;
```

Question 5. Écrire une fonction `int_pile alloue_pile()` qui crée une pile vide.

Question 6. Écrire une fonction `void libere_pile(int_pile p)` qui libère la mémoire occupée par la pile `p`.

Question 7. Écrire une fonction `int empty_pile(int_pile p)` qui renvoie un booléen et qui teste si la pile est vide.

Question 8. Écrire une fonction `void empile_pile(int_pile p, int n)` qui empile l'entier `n` sur la pile `p`.

Question 9. Écrire une fonction `int depile_pile(int_pile p)` qui dépile la pile `p` et renvoie l'élément dépilé.

Question 10. Utiliser la structure de donnée de pile pour inverser l'ordre des éléments d'une liste donnée.