TD7

Programmation en C (LC4)

Semaine du 19 mars 2012

1 Liste d'entiers triée

Soit le type :

```
typedef struct liste_entier{
  int entier;
  struct liste_entier* suivant;
} liste;
```

Exercice 1 Écrivez une fonction `liste* cree_element(int n)` qui crée un élément de liste dont l'entier contient n et le pointeur suivant `NULL`.

Solution:

```
liste* cree_element(int n){
  liste* nouvel_element = malloc(sizeof(liste));
  nouvel_element->entier = n;
  nouvel_element->suivant = NULL;
  return nouvel_element;
}
```

Exercice 2 Écrivez une fonction

```
liste* ajoute_element(liste* nouvel_element, liste* liste_courante)
```

qui ajoute l'élément de liste `nouvel_element` (créé avec la fonction écrite à l'exercice précédent) à la liste courante de manière à ce que les entiers de la liste soient triés par ordre croissant. Cette fonction renvoie un pointeur sur le premier élément de la liste ainsi modifiée.

Solution:

```
liste* ajoute_element(liste* nouvel_element, liste* liste_courante){
  //si la liste est vide
  if(liste_courante == NULL)
  return nouvel_element;
  //test sur le premier element
  if(liste_courante->entier >= nouvel_element->entier){
    nouvel_element->suivant = liste_courante;
    return nouvel_element;
  }
  //sinon, on est dans le cas general d'ajout en milieu de liste
  liste* precedent = liste_courante;
  liste* courant = liste_courante->suivant;
  while(courant != NULL && courant->entier < nouvel_element->entier){
    //remarque sur le fonctionnement du && qui si le test (courant!=NULL) est faux
    //ne fera pas le test (courant->entier < nouvel_element->entier)
    precedent = courant;
```

```
      courant = courant->suivant;
    }
    //precedent et suivant sont maintenant bien places
    precedent->suivant = nouvel_element;
    nouvel_element->suivant = courant;
  }
```

2 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 :

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

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.

Exercice 3 Écrire une fonction `struct pile_amortie *alloue_pile_amortie()` qui crée une pile amortie de capacité initiale 0.

Solution:

```
struct pile_amortie *alloue_pile_amortie()
{
  struct pile_amortie *pile = malloc(sizeof(struct pile_amortie));
  pile->occupation = 0;
  pile->capacite = 0;
  pile->elements = NULL;
  return pile;
}
```

Exercice 4 Écrire une fonction `void libere_pile_amortie(struct pile_amortie *pile)` qui libère la mémoire occupée par la pile `pile`. On veillera à bien libérer toute la mémoire occupée.

Solution:

```
void libere_pile_amortie(struct pile_amortie *pile)
{
  free(pile->elements);
  free(pile);
}
```

Exercice 5 Écrire une fonction `void empile_pile_amortie(struct pile_amortie *pile, int n)` qui empile l'entier `n` sur la pile `pile`.

Solution:

```
void empile_pile_amortie(struct pile_amortie *pile, int n)
{
  if (pile->occupation == pile->capacite) {
    if (pile->capacite == 0) {
      pile->capacite = 1;
      pile->elements = malloc(sizeof(int));
    } else {
    pile->capacite *= 2;
    pile->elements = realloc(pile->elements,
                          pile->capacite *(sizeof(int)));
    }
  }
  pile->elements[pile->occupation] = n;
  pile->occupation++;
}
```

Exercice 6 Écrire une fonction `int depile_pile_amortie(struct pile_amortie *pile)` qui dépile la pile `pile` et renvoie l'élément dépilé.

Solution:

```
int depile_pile_amortie(struct pile_amortie *pile)
{
  int n = pile->elements[pile->occupation - 1];
  pile->occupation--;
  if (pile->occupation <= pile->capacite / 4) {
    pile->capacite /= 2;
    if (pile->capacite != 0)
        pile->elements = realloc(pile->elements,
                              pile->capacite *sizeof(int));
    else { /* pile->capacite == 0 */
      free(pile->elements);
      pile->elements = NULL;
    }
  }
  return n;
}
```