

Correction TP7: Pile et file

1 Pile

Structure abstraite

Q1. Exemple de programme de test pour les piles :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include "pile.h"
5
6  int main(){
7      pile_t* p = pile_vide();
8      int x;
9
10     empiler(p, 5);
11     affiche_pile(p);
12     assert(!est_vide(p));
13
14     x = depiler(p);
15     affiche_pile(p);
16     assert(x == 5);
17
18     empiler(p, 1);
19     affiche_pile(p);
20
21     empiler(p, 2);
22     affiche_pile(p);
23
24     empiler(p, 3);
25     affiche_pile(p);
26
27     empiler(p, 4);
28     affiche_pile(p);
29
30     assert(depiler(p) == 4);
31     affiche_pile(p);
32
33     assert(depiler(p) == 3);
34     affiche_pile(p);
35
36     assert(depiler(p) == 2);
37     affiche_pile(p);
38
39     empiler(p, 5);
40     affiche_pile(p);
41
```

```

42  assert(depiler(p) == 5);
43  affiche_pile(p);
44
45  assert(depiler(p) == 1);
46  assert(est_vide(p));
47  affiche_pile(p);
48
49  // empiler 0, 1, ..., 999, puis dépile en vérifiant les valeurs
50  for (int i = 0; i < 1000; ++i){
51      empiler(p, i);
52  }
53  for (int i = 999; i >= 0; --i){
54      assert(depiler(p) == i);
55  }
56
57  free_pile(p);
58  return 0;
59  }

```

Implémentation par tableaux

Q2. Voir cours

Implémentation par liste chaînée

Voir cours.

Implémentation par tableaux redimensionnables

Q6. Modification de la structure pour prendre en compte la taille allouée :

```

1  // Invariant de structure: nb_elem <= taille_max
2  struct pile {
3      int taille_max;
4      int nb_elem;
5      int* tab;
6  };

```

Puis modification de la fonction empiler pour redimensionner lorsque le tableau est plein :

```

1  void empiler(pile_t* p, int x){
2      // Redimensionner
3      if(p->nb_elem == p->taille_max){
4          int new_taille_max = 2 * p->taille_max;
5          p->tab = realloc(p->tab, new_taille_max * sizeof(int));
6          p->taille_max = new_taille_max;
7      }
8      // Empiler
9      assert(p->nb_elem < p->taille_max);
10     p->tab[p->nb_elem] = x;
11     p->nb_elem++;
12 }

```

Pour les tests, on peut fixer la taille initiale à 10 :

```

1  pile_t* pile_vide(){
2      pile_t* p = malloc(sizeof(pile_t));
3      p->nb_elem = 0;
4      p->taille_max = 10;

```

```

5   p->tab = malloc(p->taille_max*sizeof(int));
6   return p;
7 }

```

Le programme de test est sensé empiler puis dépiler 1000 éléments : s'il s'exécute correctement, c'est que le redimensionnement fonctionne bien. On vérifie avec valgrind :

```

fredfrigo@ordi:~/TP7/pile$ valgrind ./a.out
==1120== Memcheck, a memory error detector
==1120== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1120== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==1120== Command: ./a.out
==1120==
5

1
1 2
1 2 3
1 2 3 4
1 2 3
1 2
1
1 5
1

==1120==
==1120== HEAP SUMMARY:
==1120==    in use at exit: 0 bytes in 0 blocks
==1120==   total heap usage: 10 allocs, 10 frees, 11,240 bytes allocated
==1120==
==1120== All heap blocks were freed -- no leaks are possible
==1120==
==1120== For lists of detected and suppressed errors, rerun with: -s
==1120== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Pas d'erreur ni de fuite mémoire.

```

Rétrécissement du tableau

Q7. Sur le même principe que l'empilage :

```

1  int depiler(pile_t* p){
2      assert(!est_vide(p));
3      int res = p->tab[p->nb_elem-1];
4      p->nb_elem--;
5      if (p->nb_elem <= p->taille_max/2){
6          int new_taille_max = p->taille_max/2;
7          p->tab = realloc(p->tab, new_taille_max * sizeof(int));
8          p->taille_max = new_taille_max;
9      }
10     return res;
11 }

```

Q8. Cette stratégie n'est pas efficace. En effet, supposons que l'on a une pile ayant un tableau de N cases allouées, où les N cases sont utilisées. Si l'on empile une valeur de plus, on

double la taille du tableau : $N + 1$ cases seront utilisées sur $2n$ allouées. Si l'on dépile ensuite, seules N des $2N$ cases sont utilisées, et le tableau est réalloué, on se retrouve dans la situation initiale.

Autrement dit, cette suite de 2 opérations a coûté $\Theta(N)$, et on pourrait les répéter à l'infini. Finalement, le coût amorti de l'empilage et du dépilage est en $\Theta(n)$.

- Q9.** Ce qui fait que la stratégie de redimensionnement précédente n'est pas efficace est que l'on peut enchaîner deux redimensionnements en n'ayant que peu d'opérations entre. Si l'on arrive à obliger qu'entre deux redimensionnements d'une structure à n éléments, on fait $\Theta(n)$ opérations, le coût amorti redescendra à $\Theta(1)$.

Une possibilité est de diviser la taille du tableau par 2 **lorsque le tableau n'est rempli qu'au quart** :

```
1  int depiler(pile_t* p){
2      assert(!est_vide(p));
3      int res = p->tab[p->nb_elem-1];
4      p->nb_elem--;
5      if (p->nb_elem <= p->taille_max/4){
6          int new_taille_max = p->taille_max/2;
7          p->tab = realloc(p->tab, new_taille_max * sizeof(int));
8          p->taille_max = new_taille_max;
9      }
10     return res;
11 }
```

Le contre-exemple de la question précédente ne pose plus problème. Lorsque l'on vient de causer un agrandissement d'un tableau de taille n à $2n$, il faut ensuite n opérations pour causer un autre agrandissement, ou bien $\frac{n}{2}$ opérations pour causer un rétrécissement. Inversement, après un rétrécissement, il faut aussi $\Omega(n)$ opérations pour causer un rétrécissement ou un agrandissement.

2 Application des piles

Coquille dans le sujet : l'algorithme naïf présenté est en $\mathcal{O}(n^2)$ (puisque'il y a n tours de boucles chacun en $\mathcal{O}(n)$), et pas en $\mathcal{O}(n)$.

Q10. La forme stable du polymère **abcdDCaABeEeA** est **aeA**.

Q11. On peut adapter l'algorithme de test des mots bien parenthésés :

```

Entrée(s) :  $p = p_0p_1 \dots p_{n-1}$  polymère
Sortie(s) : Forme stable de  $p$ 
1  $P \leftarrow \text{pile\_vide}()$ ;
  // Invariant:  $P$  contient les éléments de  $p_0 \dots p_{i-1}$  n'ayant pas pu réagir
2 pour  $i = 0$  à  $n - 1$  faire
3   si  $P$  non vide alors
4      $x \leftarrow \text{depiler}(P)$ ;
5     si  $x$  et  $p_i$  ne peuvent pas réagir alors
6        $\text{empiler}(P, x)$ ;  $\text{empiler}(P, s[i])$ ;
7   sinon
8      $\text{empiler}(P, s[i])$ ;
9 Renverser  $P$  dans une chaîne de caractères  $s$ , en mettant le sommet de  $P$  à la fin de  $s$ ,
  et la base de  $P$  au début de  $s$ ;
10 retourner  $s$ 

```

Q12. Ajouter une opération à la SDA revient à modifier le .h avec la spécification de l'opération, puis à l'implémenter dans le ou les .c correspondants.

Pour le .h :

```

1 /* Renvoie le nombre d'éléments de p */
2 int taille(pile_t* p);

```

Pour l'implémentation par tableau, c'est trivial : un des paramètres donne immédiatement la taille :

```

1 int taille(pile_t* p){
2     return p->nb_elem;
3 }

```

Pour celle par liste, on peut simplement rajouter un attribut qui stocke la taille, et le mettre à jour à chaque empilage/dépilage (voir fichier `pile_chaine.c` de l'archive du corrigé).

Q13. Coquille : la fonction devrait plutôt s'appeler `forme_stable`, car elle renvoie la forme stable du polymère, et pas juste la taille :

```

1 /* Renvoie la forme stable de p */
2 char* forme_stable(char* p){
3     pile_t* P = pile_vide();
4     for (int i = 0; p[i] != '\0'; ++i){
5         if (!est_vide(P)){
6             char x = depiler(P);
7             if (x - 'a' + 'A' != p[i] && x + 'a' - 'A' != p[i]){
8                 // pas de réaction
9                 empiler(P, x);
10                empiler(P, p[i]);
11            }

```

```

12     } else {
13         empiler(P, p[i]);
14     }
15 }
16 // fin des réactions: on dépile P dans une chaîne
17 int n = taille(P);
18 char* s = malloc((n+1)*sizeof(char));
19 int i = n-1;
20 while(!est_vide(P)){
21     s[i] = depiler(P);
22     i--;
23 }
24 s[n] = '\0';
25 free_pile(P);
26 return s;
27 }

```

Quelques tests :

```

1 int test(){
2     char* p = "abcd";
3     char* s = forme_stable(p);
4     assert(strcmp(s, "abcd") == 0);
5     free(s);
6
7     p = "abcdDCBA";
8     s = forme_stable(p);
9     assert(strcmp(s, "") == 0);
10    free(s);
11
12    p = "abcBbCA";
13    s = forme_stable(p);
14    assert(strcmp(s, "abA") == 0);
15    free(s);
16 }

```

Pour lancer l'algorithme sur le grand polymère donné dans l'archive, il faut commencer par lire le fichier. On ne connaît pas sa taille a priori, donc on ne sait pas combien d'espace allouer avant de commencer la lecture. Quelques solutions possibles :

- Lire une première fois le fichier, caractère par caractère, sans rien stocker, en comptant simplement le nombre de caractères lus, pour déterminer la taille du polymère, puis on sait combien d'espace allouer, et on recommence la lecture en stockant cette fois-ci les caractères lus. Inconvénient : c'est deux fois plus long que de lire le fichier une seule fois.
- Regarder, à la main, les données du fichier, soit en faisant clic-droit puis Propriétés, soit avec la commande `wc -c` qui compte le nombre d'octets d'un fichier. On voit qu'il y a 2000001 octets. Inconvénient : on ne peut pas adapter ça pour lire n'importe quel fichier, il faudrait regarder à la main à chaque nouveau fichier.

Une autre alternative est d'utiliser le système de tableau redimensionnable que l'on a vu en cours ! On alloue au départ 100 octets, puis on lit le fichier en stockant les caractères lus dans la zone allouée, que l'on agrandit à chaque fois qu'elle est pleine :

```

1 /* Renvoie l'intégralité du contenu du fichier nom_fichier. Renvoie
2    NULL si l'ouverture échoue. */
3 char* lire(char* nom_fichier){
4     FILE* f = fopen(nom_fichier, "r");
5     if (f == NULL){

```

```
6     return NULL;
7 }
8
9 char* p = malloc(101*sizeof(char));
10 int n = 0; // nombre de caractères lus
11 int n_max = 100; // nombre de caractères réservés
12 char c;
13 while (fscanf(f, "%c", &c) != EOF){
14     // réallouer si nécessaire
15     if (n == n_max){
16         n_max *= 2;
17         p = realloc(p, (2*n_max+1)*sizeof(char));
18     }
19     p[n] = c;
20     n++;
21 }
22 p[n] = '\0';
23 fclose(f);
24 return p;
25 }
```

On vérifie alors que la forme stable du grand polymère de l'archive est de taille 41451.

3 File

Implémentation par listes chaînées

Voir fichier `file/file_chaine.c` de l'archive.

Implémentation par tableaux cycliques

Voir fichier `file/file_circu.c` de l'archive. On peut modifier la valeur `Nmax` pour changer la taille maximale allouée dans l'implémentation par tableau. En prenant comme taille maximale 10, les tests de `file/test_file.c` fonctionnent, montrant que la file reste utilisable même avec plusieurs centaines d'enfilages et défilages alternés, ce qui n'est pas le cas avec l'implémentation par tableaux naïve.