Correction de l'examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 8 janvier 2025

Exercice 1 : Fonctions simples (6 points)

1.
```
/*@ requires a and b are valid addresses
    assigns *a and *b
    ensures swap the values contained in *a and *b */
void swap(char *a, char *b) {
  char tmp = *a;
  *a = *b;
  *b = tmp;
}
```

2.
```
/*@ requires t is an array of size s > 0
    assigns t[0] ... t[s-1]
    ensures rotate the values in t one step to the right */
void rotate(int t[], int s) {
  int last = t[s-1];
  /* from the end to the start, otherwise values are
     overwritten */
  for (int i = s - 1; i > 0; i -= 1)
    t[i] = t[i - 1];
  t[0] = last;
}
```

3.
```
/*@ requires n >= 0
    assigns nothing
    ensures returns a string containing n times '*' */
char *stars(int n) {
  char *r = malloc(n + 1); // sizeof (char) == 1
  for (int i = 0; i < n; i += 1)
    r[i] = '*';
  r[n] = '\0'; // strings are nul-terminated
  return r;
}
```

4.
```
/*@ requires l is a well-formed acyclic list
    assigns nothing
    ensures return the address of the last cell of l,
            or NULL if l is empty */
list last(list l) {
  if (l == NULL) return NULL;
  //@ loop invariant l != NULL
  while (l->next != NULL)
    l = l->next;
  return l;
}
```

Exercice 2 : Listes doublement chaînées (10pts)

1.
```
typedef struct cell *dl_list;
typedef struct cell cell;
struct cell {
  dl_list prev;
  int val;
  dl_list next;
};
```

   La deuxième ligne n'est pas obligatoire mais permet de ne pas devoir rajouter `struct` devant `cell` par la suite.

2.
```
/*@ requires nothing
    assigns nothing
    ensures return a doubly-linked list consisting of
            only one cell containing the value n */
dl_list singleton(int n) {
  dl_list r =  malloc(sizeof (cell));
  r->prev = NULL;
  r->next = NULL;
  r->val = n;
  return r;
}
```

   Pas de `@requires n est un entier` qui n'apporte aucune information !

3.
```
/*@ requires pl is a valid address pointing to
             an acyclic list
    assigns *pl
    ensures moves *pl to its first cell */
void go_to_first(dl_list *pl) {
  if (*pl == NULL) return;
```

```
  //@ loop invariant *pl != NULL
  while ((*pl)->prev != NULL)
    *pl = (*pl)->prev;
}
```

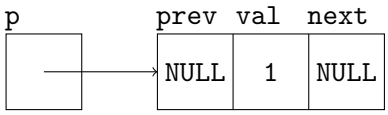
4.
```
/*@ requires pl is a valid address
    assigns *pl and its cells
    ensures add a new cell containing n to the right
            of the current position, and move to it */
void add_next(dl_list *pl, int n) {
  dl_list r = malloc(sizeof (cell));
  r->val = n;
  if (*pl == NULL) {
    *pl = r;
    r->prev = NULL;
    r->next = NULL;
    return;
  }
  r->next = (*pl)->next;
  r->prev = *pl;
  if ((*pl)->next != NULL)
    (*pl)->next->prev = r;
  (*pl)->next = r;
  *pl = r;
}
```

5.
```
/*@ requires l is a well-formed acyclic doubly-linked list
    assigns nothing
    ensures return the number of cells in l */
int length(dl_list l) {
  int r = 0;
  if (l == NULL) return 0;
  dl_list q = l->prev;
  while (q != NULL) {
    r += 1;
    q = q->prev;
  }
  while (l != NULL) {
    r += 1;
    l = l->next;
  }
  return r;
}
```

6.
```
/*@ requires l is a well-formed acyclic doubly-linked list
    assigns nothing
    ensures free all cells of l */
void free_dl_list(dl_list l) {
  go_to_first(&l);
  while (l != NULL) {
    dl_list tmp = l;
    l = l->next;
    free(tmp);
  }
}
```

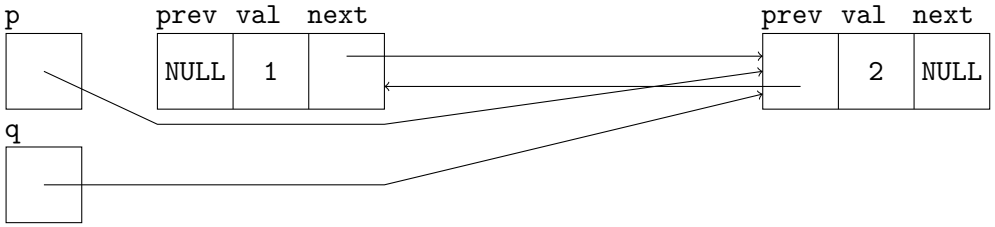
7.

Ligne 1 :

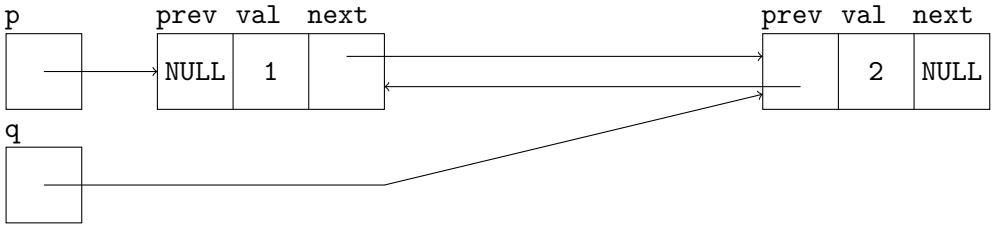


Ligne 2 :

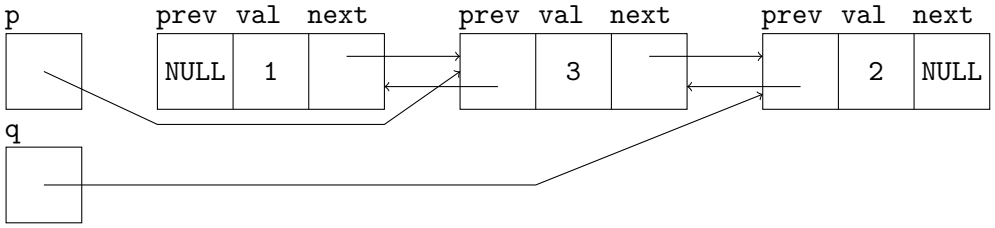
Ligne 3 :



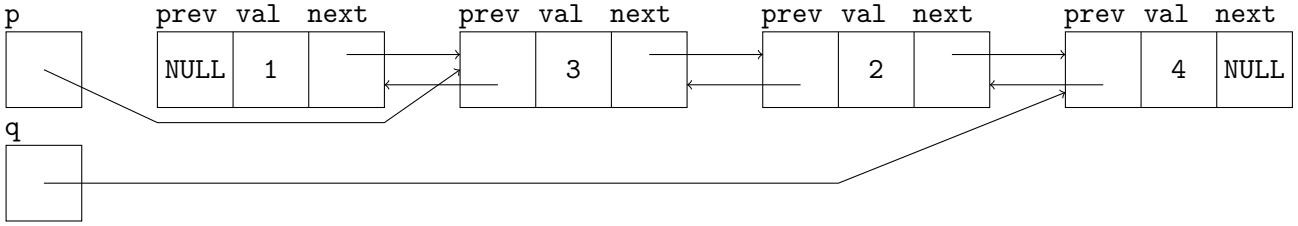
Ligne 4 :



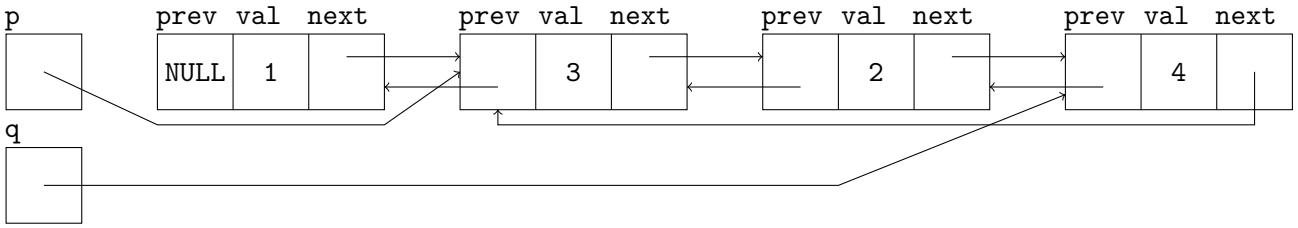
Ligne 5 :



Ligne 6 :

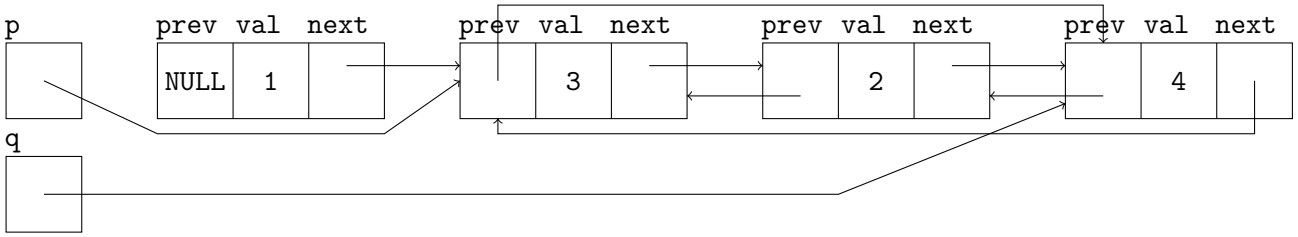


Ligne 7 :



On remarque que la liste doublement chaînée n'est plus bien formée, parce que `q->next->prev` est différent de `q`.

8.



On remarque que le maillon contenant 1 n'est plus accessible. En particulier il ne pourra plus être libéré, il y a une fuite mémoire.

On remarque également que la liste est à nouveau bien formée mais qu'elle est cyclique.

Exercice 3 : Jeu des 4 erreurs (4 points)

1. Il n'y a pas de **`return`** alors que le type de retour est **`int`**. Comme il s'agit d'une procédure, il aurait fallu mettre **`void`** comme type de retour, ou alors il aurait fallu renvoyer un entier, par exemple 0.

2. Le manque d'accolades dans le deuxième **`while`** fait que seule l'instruction `c[i] = s[i];` fait partie du corps de la boucle. Par conséquent, sauf si `s` est la chaîne vide, on boucle indéfiniment.

   Remarque 1 : le `malloc` est correct puisque **`sizeof (char)`** vaut 1.

   Remarque 2 : **`sizeof`** n'est pas une fonction, c'est un mot clef qui est utilisé par le compilateur pour calculer des tailles au moment de la compilation.

3. Il ne faut pas allouer de la mémoire pour la taille d'une liste, mais pour la taille d'un maillon. D'ailleurs, un maillon contient un champ de type `list` et il a donc forcément une taille strictement plus grande.

4. On a passé la liste par valeur et non par référence. La mémoire allouée pour le premier maillon est bien libérée, mais la liste n'avance pas sur le maillon suivant.