Correction de l'examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 8 janvier 2020

Exercice 1 : Fonctions simples

1.
```
/*@ requires x contient une adresse valide
    assigns *x
    ensures incremente *x de 1 */
void incr(int *x) {
  *x = *x + 1;
}
```

2.
```
/*@ requires i >= 0 et j >= 0 et
       t est un tableau de taille au moins max(i,j)+1
    assigns t[i] et t[j]
    ensures echange t[i] et t[j] */
void swap(double t[], int i, int j) {
  double tmp;
  tmp = t[i];
  t[i] = t[j];
  t[j] = tmp;
}
```

3.
```
/*@ requires s est une chaine de caractere correctement formee
      c'est-a-dire terminee par '\0'
    assigns nothing
    ensures compte le nombre d'occurrence de "ab" dans s */
int nb_ab(char *s) {
  int r, i;
  r = 0;
  i = 0;
  while (s[i])
    if (s[i] == 'a' && s[i+1] == 'b')
  // i+1 est une position valide dans ce cas car au pire c'est '\0'
      { r = r + 1;
        i = i + 2; // on sait que le suivant est 'b', on le passe
      }
```

```
    else
      i = i + 1;
  return r;
}
```

4.
```
/*@ requires t est un tableau de taille s
    assigns nothing
    ensures retourne une copie de t dans laquelle
    l'ordre des elements est inverse */
int *reversed_copy(int *t, int s) {
  int *r;
  int i;
  r = malloc(s * sizeof(int));
  for (i = 0; i < s; i = i + 1)
    r[i] = t[s - 1 - i];
  return r;
}
```

Exercice 2 : Arbres

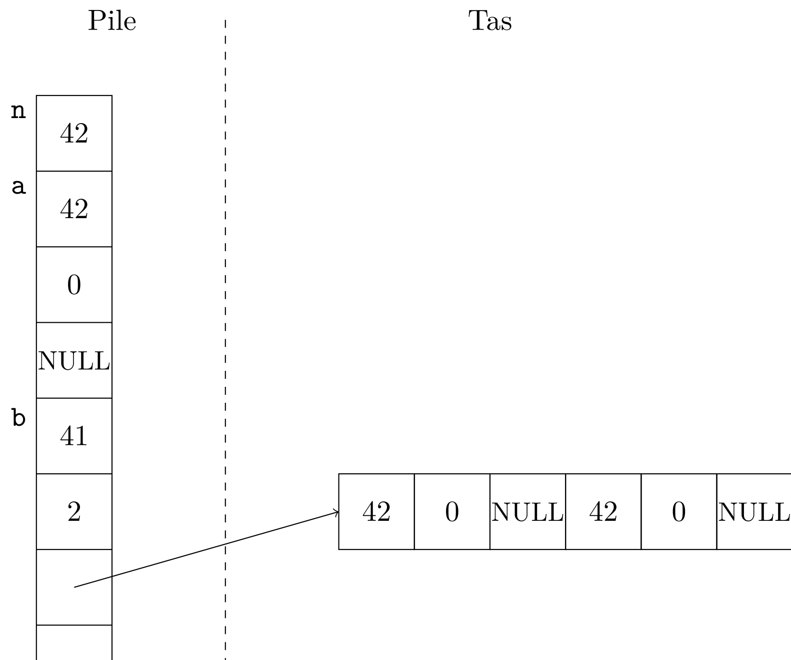
1.
```
/*@ requires t est bien forme
      c'est-a-dire que t.children.nodes est de taille
      t.children.nb_nodes, et tous les elements de
      t.children.nodes sont eux aussi bien formes
    assigns nothing
    ensures affiche t sur la sortie standard */
void print_tree(tree t) {
  int i;
  printf("%d", t.val);
  if (t.children.nb_nodes > 0) {
    printf("␣(");
    for (i = 0; i < t.children.nb_nodes - 1; i++) {
      print_tree (t.children.nodes[i]);
      printf(",␣");
    }
    print_tree (t.children.nodes[t.children.nb_nodes - 1]);
    printf(")");
  }
}
```

2.
```
/*@ requires nothing
    assign nothing
    ensures retourne une feuille contenant e */
tree leaf(int e) {
  tree r;
```

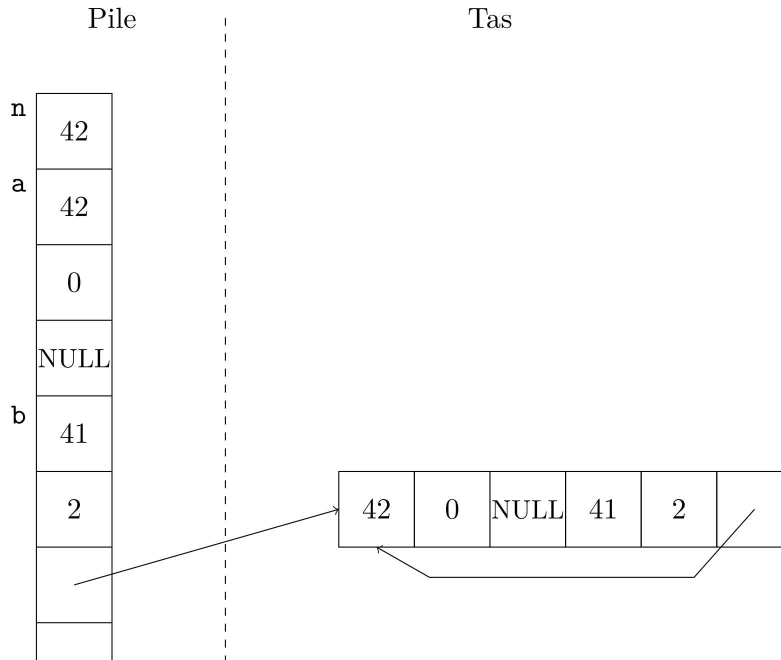
```
    r.val = e;
    r.children.nb_nodes = 0;
    /* il n'est pas obligatoire d'initialiser
       r.children.nodes car il ne devrait pas etre accede */
    return r;
  }
```

3.
```
/*@ requires left et right sont bien formes
    assign nothing
    ensures retourne un arbre avec i a la racine
            et left et right comme enfants */
tree binary_node(int e, tree left, tree right) {
  tree r;
  r.val = e;
  r.children.nb_nodes = 2;
  r.children.nodes = malloc(2 * sizeof(tree));
  r.children.nodes[0] = left;
  r.children.nodes[1] = right;
  return r;
}
```

4. Un `tree` contient deux champs : un `int` et un `forest`. Le `forest` est lui même composé de deux champs, un `int` et un `tree *`. Un `tree` sera donc sur trois cases mémoires, deux de taille `int` et une de taille `tree *`. Cette dernière contiendra des adresses.
   Avant la ligne 6 on a l'état suivant :



   Juste après la ligne 6 on aura :



On remarque que `b` est maintenant cyclique. On appliquant par exemple `print_tree` dessus, le programme ne s'arrêterait pas.

5. ```
/*@ requires f est bien formee
    assigns nothing
    ensures retourne une copie de f dans laquelle
    l'ordre des elements est inverse */
forest inversed_copy(forest f) {
  forest r;
  int i;
  r.nb_nodes = f.nb_nodes;
  r.nodes = malloc(f.nb_nodes * sizeof(tree));
  for (i = 0; i < f.nb_nodes; i = i + 1)
    r.nodes[i] = f.nodes[f.nb_nodes - 1 - i];
  return r;
}
```

6. ```
/*@ requires t adresse valide et *t bien forme
    assigns *t et ses descendants
    ensures transforme *t en son image dans le miroir */
void mirror(tree *t) {
  if (t->children.nb_nodes != 0) {
    int i;
    tree *tmp = t->children.nodes;  // pour le free
    t->children = inversed_copy(t->children);
    free(tmp);
    for (i = 0; i < t->children.nb_nodes; i++)
      mirror(&(t->children.nodes[i]));
```

```
    }
}
```

Exercice 3 : Étude de fonctions

1. La deuxième partie de la condition de la boucle est une affectation (simple `=`) et non pas un test d'égalité (double `==`). Par conséquent elle va modifier le contenu du tableau et retourner la valeur qui a été affectée.

   On va donc affecter `t[1]` c'est-à-dire `1` dans `t[0]` et continuer la boucle ; puis `t[2]` c'est-à-dire `1` dans `t[1]` et continuer la boucle ; puis `t[3]` c'est-à-dire `0` dans `t[2]` et arrêter la boucle (condition fausse).

   Au final on renvoie donc `0` car `i` qui vaut `2` est plus petit que 4, et le tableau contient maintenant 1, 1, 0, 0, 1.

2. Comme il n'y a pas d'accolades après le **`while`**, seule l'instruction `i = i + 1` est exécutée, la condition de la boucle est toujours vraie et le programme s'exécute jusqu'à ce que le Soleil, devenu une géante rouge, englobe la Terre (ou, plus probablement, jusqu'à ce que quelqu'un interrompe le programme ou éteigne l'ordinateur).

3. La fonction `remove_head` prend `l` par valeur et non par référence. `l` ne sera donc pas modifiée. Par contre, le maillon pointé par `l` est lui bien libéré par l'instruction `free(tmp)`.

   Par conséquent, lors de `l->val`, on essaie d'accéder à un maillon qui n'est plus alloué. On aura dans le meilleur des cas une erreur de segmentation, dans le pire des cas l'affichage d'une valeur qui peut ou non être égale à 1.