Correction de l'examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 24 janvier 2024

Exercice 1 : Fonctions simples (6 points)

1.
```
/*@requires px is a valid address
  @assigns *px
  @ensures double the content of *px */
void double_variable(int *px) {
  *px = 2 * *px;
}
```

2.
```
/*@requires t is an array of size s > 0
  @assigns nothing
  @ensures return an index of the maximum value of t */
int maximum_index(double t[], int s) {
  int r = 0;
  for (int i = 1; i < s; i +=1)
    if (t[i] > t[r])
      r = i;
  return r;
}
```

3.
```
/*@requires n > 0
  @assigns nothing
  @ensures return an array containing the n first multiples of m */
int *multiples(int n, int m) {
  int *res = malloc(n * sizeof (int));
  for (int i = 0; i < n; i += 1)
    res[i] = i * m;
  return res;
}
```

4.
```
/*@requires s is a nul-terminated string
  @assigns nothing
  @ensures return the number of spaces in s */
int spaces(char *s) {
  int i = 0;
  int r = 0;
  while (s[i] != '\0') {
```

```
    if (s[i] == ' ')
      r += 1;
    i += 1;
  }
  return r;
}
```

5. ```
   /*@requires s is a nul-terminated string
     @assigns nothing
     @ensures test if s is a palindrome */
   int palindrome(char *s) {
     int j = 0;
     while (s[j] != '\0')
       j += 1;
     j -= 1;
     int i = 0;
     while (i < j) {
       if (s[i] != s[j])
         return 0;
       i += 1;
       j -= 1;
     }
     return 1;
   }
   ```

6. ```
   /*@requires l is a well-formed acyclic list
     @assigns nothing
     @ensures return the number of even elements in l */
   int even_number(list l) {
     int r = 0;
     while (l != NULL) {
       if (l->val % 2 == 0)
         r += 1;
       l = l->next;
     }
     return r;
   }
   ```

Exercice 2 : Arbres (10pts)

1. ```
   typedef struct node *tree;
   struct node {
     int val;
     tree child;
     tree sibling;
   ```
   ```

```
};
```

2.
```
/*@requires a and b are not empty, b->sibling is NULL
  @assigns a->child, b->sibling;
  @ensures put b as the leftmost child of a */
void add_left(tree a, tree b) {
  b->sibling = a->child;
  a->child = b;
}
```

3.
```
/*@requires a and b are not empty
  @assigns one of the children of a
  @ensures put b as the rightmost child of a */
void add_right(tree a, tree b) {
  if (a->child == NULL) {
    a->child = b;
    return;
  }
  tree r = a->child;
  while (r->sibling != NULL)
    r = r->sibling;
  // At the end of the loop, r contains the rightmost child of a
  r->sibling = b;
}
```

4.
```
/*@requires nothing
  @assigns nothing
  @ensures return a newly allocated tree containing only
    one node with the value n */
tree create_leaf(int n) {
  tree res = malloc(sizeof (struct node));
  res->val = n;
  res->child = NULL;
  res->sibling = NULL;
  return res;
}
```

5.
```
/*@requires t is an array of size s
    for all i st 0 <= i < s, t[i] is a non-empty tree
      and t[i]->sibling == NULL
  @assigns t[i]->sibling for i st 0 <= i < s - 1
  @ensures returns a tree whose root is a newly allocated node
    and whose children are t[i] for i st 0 <= i < s */
tree create_node(int n, tree t[], int s) {
  tree res = malloc(sizeof (struct node));
  res->val = n;
```

```
  res->sibling = NULL;
  res->child = t[0];
  for (int i = 0; i < s - 1; i += 1) {
    t[i]->sibling = t[i+1];
  }
  return res;
}
```

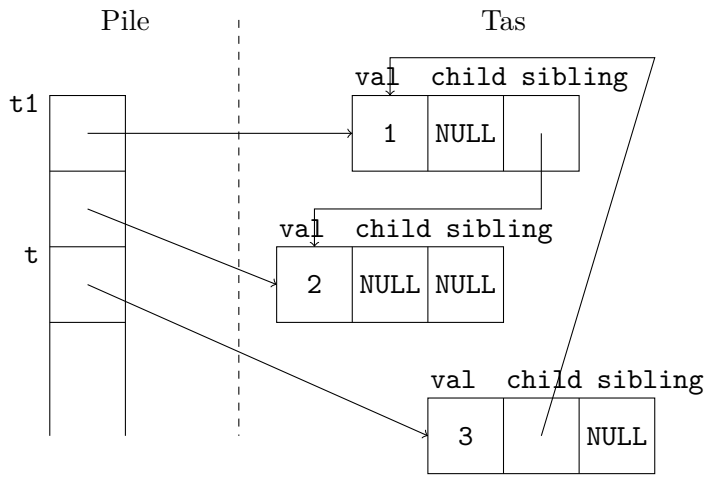
6. ```
   /*@requires nothing
     @assigns nothing
     @ensures free all memory allocated to build t */
   void free_tree(tree t) {
     if (t == NULL) return;
     free_tree(t->child);
     free_tree(t->sibling);
     free(t);
   }
   ```

7. ```
   /*@requires nothing
     @assigns nothing
     @ensures print t in a prefix fashion */
   void print_tree(tree t) {
     if (t == NULL) return;
     printf("%d ", t->val);
     if (t->child != NULL) {
       printf("[ ");
     print_tree(t->child);
     printf("] ");
     }
     print_tree(t->sibling);
   }
   ```

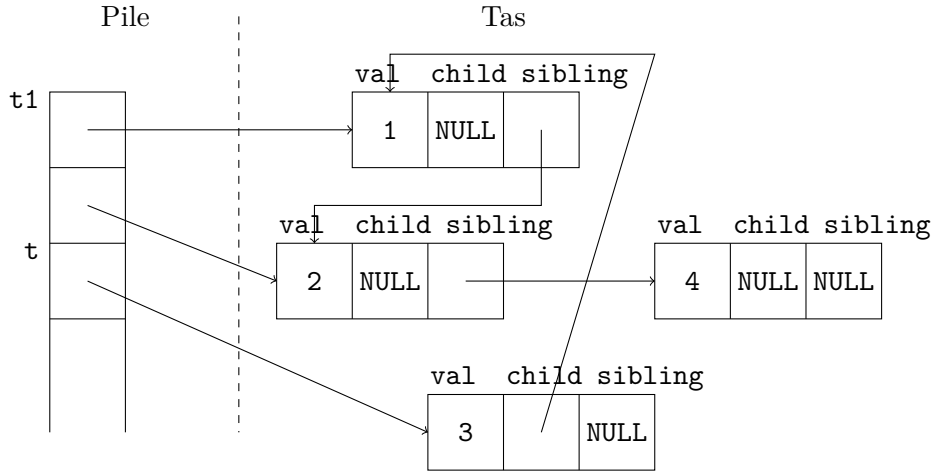
   Ou alors

   ```
   void print_tree(tree t) {
     while (t != NULL) {
       printf("%d ", t->val);
       if (t->child != NULL) {
         printf("[ ");
         print_tree(t->child);
         printf("] ");
       }
       t = t->sibling;
     }
   }
   ```

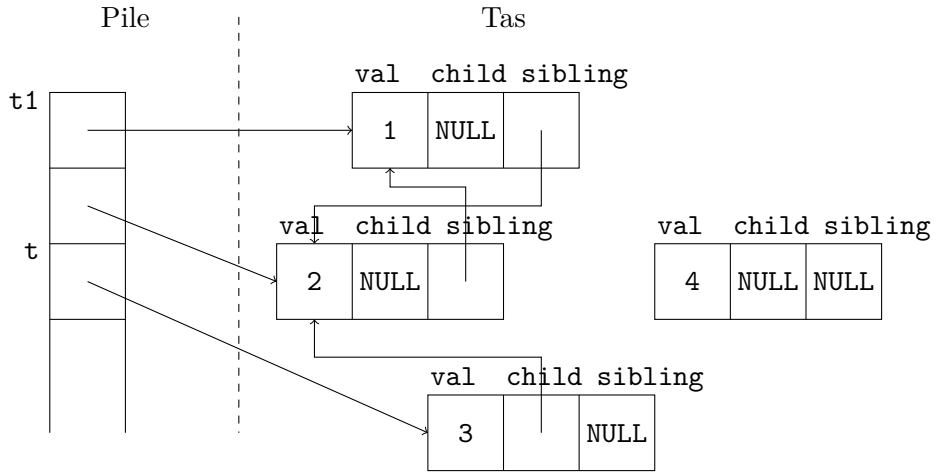
8. Après la ligne 4 :



Après la ligne 5 :



Après la ligne 6 :



On remarque que :

a) Il y a une fuite mémoire au niveau du nœud contenant 4 qui n'est plus accessible et n'a pas été libéré.

b) L'arbre pointé par `t` est devenu cyclique.

c) L'appel à `print_tree(t)` ligne 7 affiche `3 [2 1 2 1 2 1`... et boucle indéfiniment (deuxième version de `print_tree`) ou fait un dépassement de capacité de la pile (*stack overflow*, première version de `print_tree`).

9.
```
/*@requires t is a valid address
  @assigns all nodes of *t
  @ensures replace *t by its mirror image */
void mirror(tree *t) {
  if (*t == NULL) return;
  tree prec = NULL;
  tree curr = *t;
  while (curr != NULL) {
    mirror(&curr->child);
    tree tmp = curr->sibling;
    curr->sibling = prec;
    prec = curr;
    curr = tmp;
  }
  *t = prec;
}
```

Exercice 3 : Jeu des 4 erreurs (4 points)

1. Malgré l'indentation, `l = l->next;` ne fait pas partie du corps de la boucle. Il mettre une accolade ouvrante après `while (l != NULL)` et une fermante après `l = l->next;`.
2. Comme `n` est passée par valeur, seule une copie est modifiée. Il faut passer `n` par référence en remplaçant chaque `n` par `*n`.
3. Il n'y a pas assez de mémoire allouée si $s > 1$. Il faut faire `malloc(s * sizeof (double))`.
4. À cause des arrondis sur les nombres à virgules flottantes, `0 + 0.1 + 0.1 + 0.1` n'est pas égal à `0.3`. Par conséquent, la boucle ne s'arrêtera jamais.