Examen final de programmation impérative

ÉNSIIE, semestre 1

mercredi 24 janvier 2024

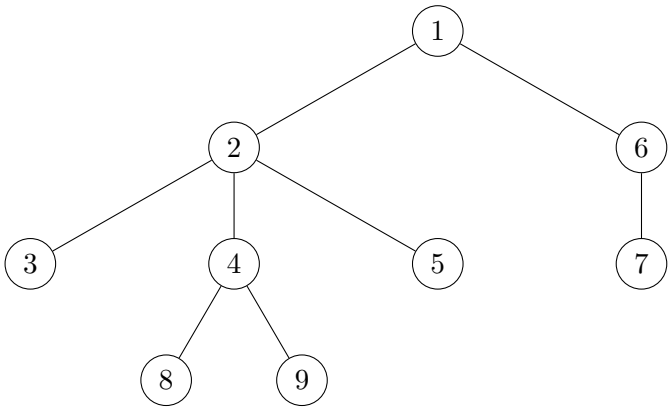
Durée : 1h45.
Tout document papier autorisé. Aucun appareil électronique autorisé (sauf dérogation par la scolarité).
Ce sujet comporte 3 exercices indépendants, qui peuvent être traités dans l'ordre voulu. Il contient 5 pages. Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points. Il va de soi que toute réponse devra être justifiée, et que toute fonction devra être commentée (au minimum `require`, `assigns` et `ensures`).

Exercice 1 : Fonctions simples (6 points)

1. Proposer une procédure dont l'effet est de doubler la valeur d'une variable de type `int` définie auparavant. (Exemple de cas d'utilisation : la fonction est définie *avant* `main`. À l'intérieur de `main`, une variables est définie et la fonction appelée avec cette variable. On n'écrira pas la fonction `main`.)
2. Proposer une fonction qui prend en paramètre un tableau non-vide de `double` et sa taille et qui retourne l'indice d'une case contenant la valeur maximum du tableau.
3. Proposer une fonction qui prend en paramètres deux entiers n et m et qui retourne un tableau contenant les n premiers multiples de m en partant de 0.
4. Proposer une fonction qui prend en paramètre une chaîne de caractères et qui retourne le nombre d'espaces dans la chaîne.
5. Proposer une fonction qui prend en paramètre une chaîne de caractères et qui teste si cette chaîne est un palindrome, c'est-à-dire qu'elle se lit de la même façon de gauche à droite ou de droite à gauche, comme par exemple `"radar"` ou `"Roma summus amoR"`.
6. Proposer une fonction qui prend en paramètre une liste chaînée sur des `int`, et qui retourne le nombre d'entiers pairs contenus dans la liste.

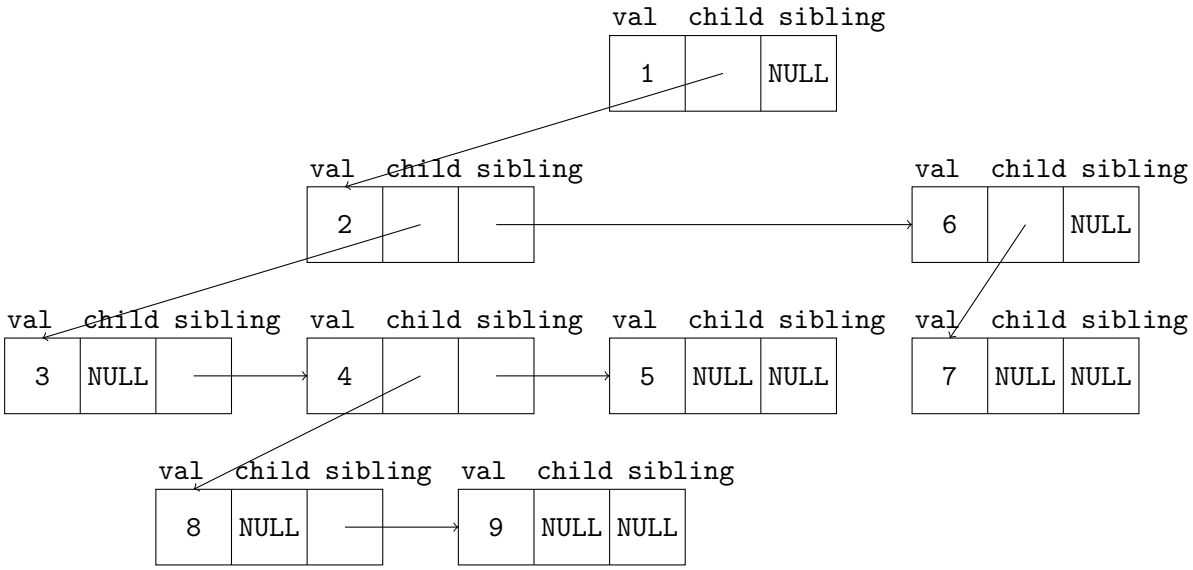
Exercice 2 : Arbres (10pts)

On considère des arbres dont les nœuds contiennent des entiers et qui peuvent avoir un nombre quelconque de fils, comme par exemple :

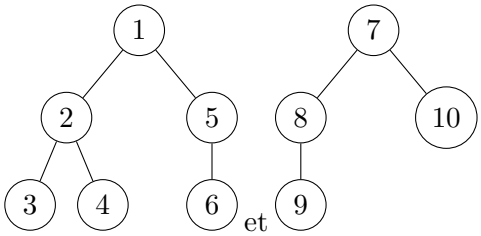


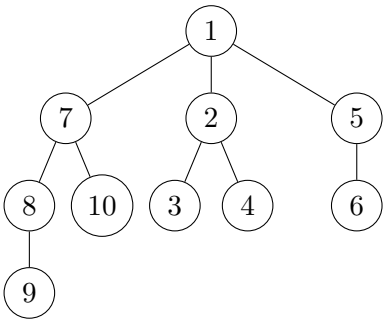
Une façon de représenter ces arbres est d'utiliser une généralisation des listes chaînées, dans laquelle les maillons (qui correspondront alors en fait aux nœuds de l'arbre) auront trois champs : `val` contenant la valeur entière, `child` pointant sur le fils le plus à gauche, et `sibling` pointant sur le frère immédiatement à droite. Comme pour les listes, si le frère immédiatement à droite n'existe pas (resp. s'il n'y a pas de fils), alors le champ `sibling` (resp. `child`) vaudra `NULL`. En particulier, à la racine, on supposera que le champ `sibling` est `NULL`.

L'exemple ci-dessus sera donc représenté par :

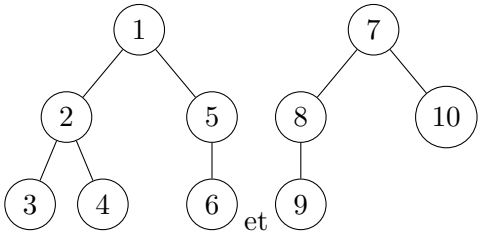


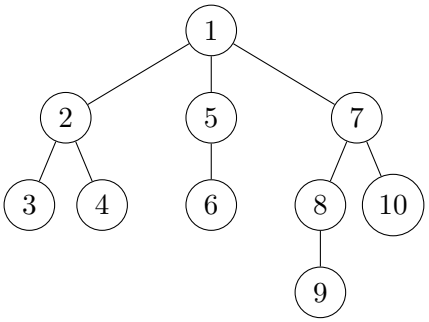
1. Définir le type `tree` des arbres comme un pointeur vers un type `node`, ce type `node` étant défini comme un enregistrement contenant les trois champs `val`, `child` et `sibling`. (Vous écrirez aussi la définition de `node`.)
2. Écrire une procédure `add_left` qui prend en paramètre deux arbres et qui ajoute le deuxième comme fils le plus à gauche du premier. Ainsi, avec les arbres



et le premier arbre va devenir


3. Écrire une procédure `add_right` qui prend en paramètre deux arbres et qui ajoute le deuxième comme fils le plus à droite du premier. Ainsi, avec les arbres



et le premier arbre va devenir


4. Écrire une fonction `create_leaf` qui prend en paramètre un entier n qui retourne un arbre qui ne contient qu'un seul nœud dont la valeur est n.
5. Écrire une fonction `create_node` qui prend trois paramètres : un entier n, ainsi qu'un tableau d'arbres et sa taille s supposée strictement positive ; `create_node` retourne un arbre dont la racine est un nœud nouvellement créé qui contient l'entier n et dont les fils sont les arbres contenus dans le tableau.
6. Écrire une procédure `free_tree` qui prend en paramètre un arbre et qui libère toute la mémoire allouée dans cet arbre.

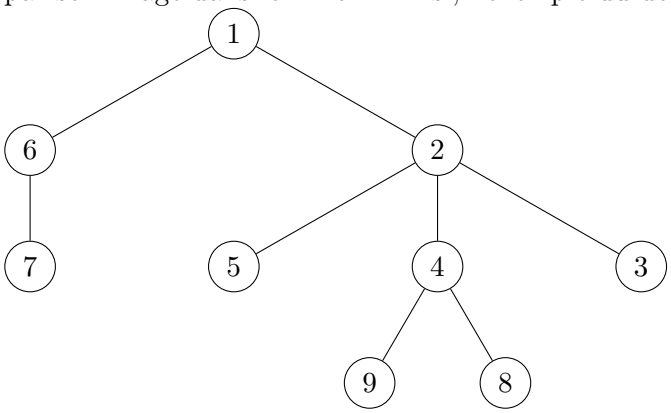
   Indication : faire des appels récursifs.
7. Écrire une procédure `print_tree` qui prend en paramètre un arbre et qui l'affiche suivant un ordre préfixe en profondeur d'abord : on affiche d'abord la valeur, puis s'il y a des enfants on les affiche récursivement entre deux crochets, puis on affiche son frère s'il existe.

   Sur l'exemple du début, on affichera donc : `1 [ 2 [ 3 4 [ 8 9 ] 5 ] 6 [ 7 ] ]`.
8. Soit le programme suivant :

```
1  tree t1[2];
2  t1[0] = create_leaf(1);
3  t1[1] = create_leaf(2);
4  tree t = create_node(3, t1, 2);
5  add_right(t, create_leaf(4));
6  add_left(t, t1[1]);
7  print_tree(t);
```

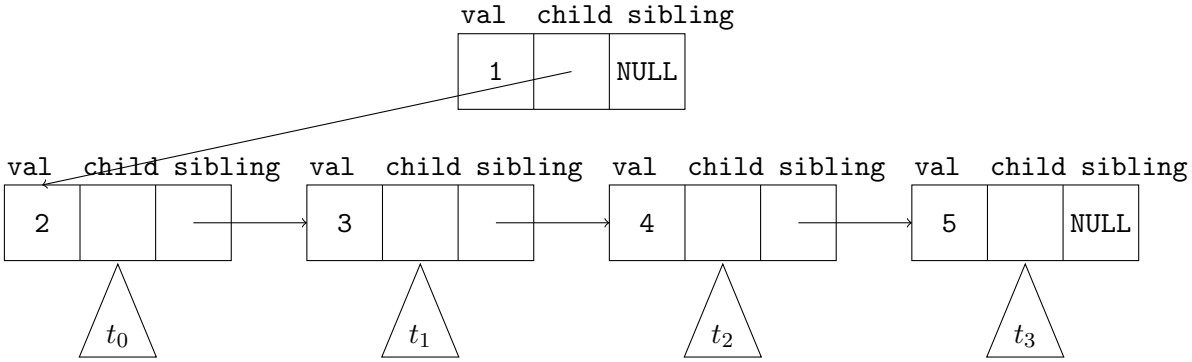
   Représenter l'état de la mémoire au cours de son exécution. Que remarque-t-on (3 remarques à faire) ?

9. Écrire une procédure `mirror` qui prend un arbre **par référence** et qui le remplace par son image dans le miroir. Ainsi, l'exemple du début de l'exercice deviendrait :

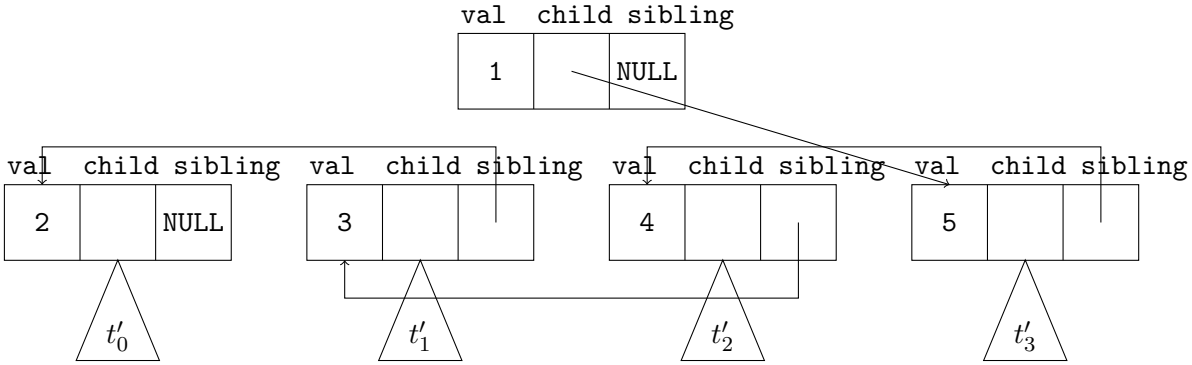


Pour cela, l'idée est de faire des appels récursifs sur les enfants. Par conséquent, la « racine » du paramètre pourra avoir des frères, et il s'agira d'inverser la chaîne constituée par ces frères, comme on inverse une liste chaînée.

Par exemple, si on a



on aura au final



où t'_i est obtenu en appliquant récursivement `mirror` sur t_i.

Exercice 3 : Jeu des 4 erreurs (4 points)

Les procédures et fonctions suivantes ne sont pas correctes par rapport à la spécification donnée. Les corriger en expliquant pourquoi il y a une erreur. Le cas échéant, on suppose

que les bons **#include** ont été écrits.

1. On considère la définition des listes chaînées vue en cours.

```
/*@requires l is an acyclic linked list
  @assigns nothing
  @ensures print the content of l */
void print_list(list l) {
  while (l != NULL)
    printf("[%d] -> ", l->val);
    l = l->next;
  printf("[]");
}
```

2.
```
/*@requires n is not the smallest reprensentable int
  @assigns n
  @ensures replace n by its absolute value */
void put_abs(int n) {
  if (n < 0) n = -n;
}
```

3.
```
/*@requires s > 0
  @assigns nothing
  @ensures return a newly allocated array containing
    s times the value n */
double *create_array(double n, int s) {
  double *res = malloc(sizeof (double));
  for (int i = 0; i < s; i += 1)
    res[i] = n;
  return res;
}
```

4.
```
/*@requires nothing
  @assigns nothing
  @ensures print consecutive values from 0 to 0.3
     by steps of 0.1 */
void print_values() {
  double i = 0;
  do {
    printf("%g ", i);
    i += 0.1;
  } while (i != 0.3);
}
```