DS 4

Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé dans la question ou le titre de la section.

Questions de cours

Question de cours 1 : Définir le principe de mémoïzation et donner un exemple (on n'écrira pas de programme). L'ensemble n'a pas besoin de dépasser dix lignes.

Question de cours 2 : Pourquoi la programmation dynamique n'a-t-elle pas de pertinence si le problème auquel on fait face revient à calculer une fonction récursive dont chaque appel n'en déclenche qu'un autre ?

Question de cours 3 : Donner la différence entre les notions d'élément minimal et de plus petit élément dans un ensemble ordonné.

Question de cours 4 : Donner la définition de l'ordre structurel, exemple à l'appui (dans un langage précis ou non, au choix).

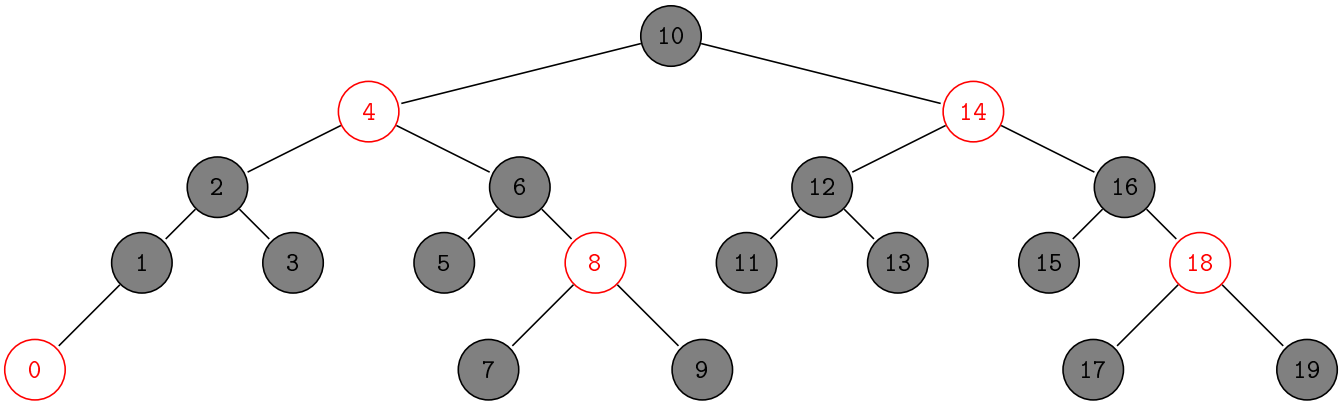
Question de cours 5 : Définir la notion d'ordre bien fondé, exemple à l'appui.

Question de cours 6 : Créer une structure en C permettant de réaliser des arbres d'entiers d'arité quelconque. Créer alors une instance de cette structure ayant une portée au choix (mais précisée) dont la racine a trois enfants et la taille est au moins de cinq.

Question de cours 7 : Définir la notion d'arbre binaire presque complet. En profiter pour prouver que tous les sous-arbres d'un arbre binaire presque complet sont des arbres binaires presque complets et donner en la justifiant une relation entre la hauteur et la taille d'un arbre binaire presque complet (préciser d'abord de manière bien visible la convention choisie pour la hauteur).

Question de cours 8 : Montrer que le parcours en profondeur infixe d'un ABR donne une séquence croissante.

Question de cours 9 : On donne l'arbre bicolore ci-après (les nœuds noirs ont leur fond grisé). Exécuter une opération de suppression (il n'y a pas unicité, et tant que l'arbre obtenu est encore bicolore et que le processus est bien expliqué tout est en ordre) d'une valeur successivement pour les valeurs 0, 7, 8, 9, 14 et 10. Programme et méthode figurent aux deux pages suivantes.




```
let rotation_droite a = match a with
| Vide -> failwith "Arbre vide"
| Noeud(Vide, _, _) -> failwith "Fils gauche vide"
| Noeud(Noeud(a1, n2, a2), n1, a3) -> Noeud(a1, n2, Noeud(a2, n1, a3));;
```

```
let rotation_gauche a = match a with
| Vide -> failwith "Arbre vide"
| Noeud(_, _, Vide) -> failwith "Fils droit vide"
| Noeud(a1, n2, Noeud(a2, n1, a3)) -> Noeud(Noeud(a1, n2, a2), n1, a3);;

let rec minimum_bicolore a = match a with
| Vide -> failwith "Cas impossible"
| Noeud(Vide, (x, _), _) -> x
| Noeud(g, _, _) -> minimum_bicolore g;;

let pas_rouge a = match a with
| Noeud(_, (_, false), _) -> false
| _ -> true;;

let rec corrige_deficit_gauche a = match a with
| Noeud(_, _, Noeud(Noeud(dgg, (dgx, true), dgd), (dx, false), Noeud(ddg, (ddx, true), ddd))) ->
    (match rotation_gauche a with
    | Noeud(Noeud(gg, (gx, true), gd), (dx, false), d) ->
    let gbis, defbis = corrige_deficit_gauche(Noeud(gg, (gx, false), gd)) in Noeud(gbis, (dx, true), d), defbis
    | _ -> failwith "Cas impossible")
| Noeud(g, (x, coul), Noeud(dg, (dx, true), dd)) when pas_rouge dg && pas_rouge dd ->
    Noeud(g, (x, true), Noeud(dg, (dx, false), dd)), if coul then 1 else 0
| Noeud(g, xc, Noeud(Noeud(dgg, (dgx, false), dgd), (dx, true), Noeud(ddg, (ddx, true), ddd))) ->
    corrige_deficit_gauche(Noeud(g, xc,
      rotation_droite(Noeud(Noeud(dgg, (dgx, true), dgd), (dx, false), Noeud(ddg, (ddx, true), ddd)))
    ))
| Noeud(g, (x, couleur), Noeud(dg, (dx, true), Noeud(ddg, (ddx, false), ddd))) ->
  begin
    match rotation_gauche(a) with
    | Noeud(Noeud(g, (x, couleur), dg), (dx, dcouleur), Noeud(ddg, (ddx, ddcouleur), ddd)) ->
        Noeud(Noeud(g, (x, true), dg), (dx, couleur), Noeud(ddg, (ddx, true), ddd)), 0
    | _ -> failwith "Ce cas ne se produit pas"
  end
| _ -> failwith "Cas impossible";;

let rec corrige_deficit_droite a = match a with
| Noeud(Noeud(Noeud(ggg, (ggx, true), ggd), (gx, false), Noeud(gdg, (gdx, true), gdd)), _, _) ->
    (match rotation_droite a with
    | Noeud(g, (gx, false), Noeud(dg, (dx, true), dd)) ->
    let dbis, defbis = corrige_deficit_droite(Noeud(dg, (dx, false), dd)) in Noeud(g, (gx, true), dbis), defbis
    | _ -> failwith "Cas impossible")
| Noeud(Noeud(gg, (gx, true), gd), (x, coul), d) when pas_rouge gg && pas_rouge gd ->
    Noeud(Noeud(gg, (gx, false), gd), (x, true), d), if coul then 1 else 0
| Noeud(Noeud(Noeud(ggg, (ggx, true), ggd), (gx, true) , Noeud(gdg, (gdx, false), gdd)) , xc, d) ->
    corrige_deficit_droite(Noeud(
      rotation_gauche(Noeud(Noeud(ggg, (ggx, true), ggd), (gx, false), Noeud(gdg, (gdx, true), gdd)))
    , xc, d))
| Noeud(Noeud(Noeud(ggg, (ggx, false), ggd), (gx, true), gd), (x, couleur), d) ->
  begin
    match rotation_droite(a) with
    | Noeud(Noeud(ggg, (ggx, ggcouleur), ggd), (gx, gcouleur), Noeud(gd, (x, couleur), d)) ->
        Noeud(Noeud(ggg, (ggx, true), ggd), (gx, couleur), Noeud(gd, (x, true), d)), 0
    | _ -> failwith "Ce cas ne se produit pas"
  end
| _ -> failwith "Cas impossible";;
```

```
let rec suppression_bicolore_aux elt a = match a with
| Vide -> failwith "Introuvable"
| Noeud(g, (x, c), d) when elt < x -> let gbis, deficit = suppression_bicolore_aux elt g in
if deficit = 0 then Noeud(gbis, (x, c), d), 0 else corrige_deficit_gauche (Noeud(gbis, (x, c), d))
| Noeud(g, (x, c), d) when elt > x -> let dbis, deficit = suppression_bicolore_aux elt d in
if deficit = 0 then Noeud(g, (x, c), dbis), 0 else corrige_deficit_droite (Noeud(g, (x, c), dbis))
| Noeud(g, (_, false), Vide) -> g, 0
| Noeud(Vide, (_, false), d) -> d, 0
| Noeud(Noeud(gg, (gx, false), gd), (_, true), Vide) -> Noeud(gg, (gx, true), gd), 0
| Noeud(Vide, (_, true), Noeud(dg, (dx, false), dd)) -> Noeud(dg, (dx, true), dd), 0
| Noeud(Vide, (_, true), Vide) -> Vide, 1
| Noeud(g, (_, c), d) -> let m = minimum_bicolore d in let dbis, deficit = suppression_bicolore_aux m d in
if deficit = 0 then Noeud(g, (m, c), dbis), 0 else corrige_deficit_droite (Noeud(g, (m, c), dbis));;

let suppression_bicolore elt a = match suppression_bicolore_aux elt a with
| Vide, _ -> Vide
| Noeud(g, (x, _), d), _ -> Noeud(g, (x, true), d);;
```

Voici un résumé de la méthode de suppression (pas exactement identique au programme écrit mais les deux fonctionnent) :

- On procède à une suppression comme dans un ABR, avec l'éventuel échange du nœud à supprimer avec le minimum du sous-arbre enraciné en ce nœud, les couleurs étant maintenues aux positions en question.
- La position où la suppression a lieu est désormais une position où au moins un fils est vide. La suppression a lieu et :
  - S'il y a un fils non vide (on rappelle que l'autre est alors vide), il est forcément rouge. Il prend alors la place du nœud supprimé, en devenant noir (car le nœud supprimé l'était forcément), puis la suppression est terminée.
  - S'il y a deux fils vides et que le nœud supprimé était rouge, c'est fini.
  - Sinon, le vide laissé fait baisser la hauteur noire de la branche en cours, ce qui peut être problématique. On va remonter cette branche en déplaçant le problème jusqu'à ce qu'il soit réglé, voir ci-après.

Parlons de la réparation de la modification de la hauteur noire d'une branche dans une nouvelle distinction de cas sur le nœud problématique, qui est initialement le vide laissé par la suppression et qu'on notera n :

- Si n est la racine, on considère que le problème est résolu, c'est l'arbre entier qui a perdu une unité de hauteur noire.
- Si n a un père, noté p, ayant un autre fils (forcément, en raison de la hauteur noire précédente) noté f, on étudie la couleur de f :
  - Si f est rouge, ce qui garantit que ses enfants éventuels sont tous noirs et que p l'est aussi, alors on fait une rotation gauche ou droite sur le sous-arbre enraciné en p suivant le fait que f soit le fils droit ou gauche de p, respectivement, et on échange les couleurs de f et p. On passe alors aux sous-cas suivants en adaptant les notations à la nouvelle situation. [1]
  - Si f est noir avec deux fils noirs, on met f en rouge et on déplace le problème à p dans un nouveau tour de boucle.
  - Sinon, si f est noir avec son fils droit noir, on met f en rouge, son fils gauche en noir, on fait une rotation droite dans le sous-arbre enraciné en f et on passe à l'étape suivante avec les notations adaptées.
  - Sinon, f est noir avec son fils droit rouge (peu importe son fils gauche), on fait alors une rotation gauche sur le sous-arbre enraciné en p. On met f à la couleur de p, on met p et le fils droit de f en noir, et le problème est résolu, ce qui permet là aussi de terminer la suppression.

1. **Pour ces sous-cas, on évitera d'alourdir les choses et on supposera que f est le fils droit de p. Dans le cas contraire, on changera toutes les occurrences de « droit(e) » par « gauche » et inversement, pour les fils comme pour les rotations.**

Exercices surtout théoriques dans un langage au choix

Exercice $\alpha 1$: On considère un tableau de `n` tableaux d'entiers tous de même taille `m`. Un chemin dans ce tableau est une séquence de couples d'indices partant de `(0, 0)` et progressant jusqu'à `(n-1, m-1)` en augmentant à chaque étape l'un des deux indices d'un. La valeur d'un tel chemin est la somme des entiers rencontrés tout le long du chemin en faisant la double indexation intuitive. Écrire une fonction qui prend en entrée un tel tableau (ainsi que les deux tailles si c'est fait en C) et qui calcule la valeur maximale d'un chemin sur ce tableau. Calculer la complexité de cette fonction. Il faut que cette complexité soit linéaire en la taille de l'entrée pour que la fonction soit acceptée. Préciser ensuite comment adapter la fonction pour que la fonction renvoie plutôt le chemin en tant que séquence de couples d'indices (type au choix). On pourra baliser des portions de code et écrire la modification ou l'ajout suggérés.

Exercice $\alpha 2$: Même exercice (sauf la partie revenant à construire un chemin réalisant le maximum) mais en considérant la somme maximale d'une branche dans un arbre d'arité quelconque.

Exercices en OCaml

Exercice $\beta 1$: Écrire deux types pour représenter respectivement des arbres binaires et des arbres d'arité quelconque. Rappeler une transformation pour passer d'un arbre en arbre binaire (un dessin annoté peut suffire). Écrire cette transformation et la transformation inverse.

Exercice $\beta 2$: Écrire une fonction qui prend en argument une liste `l` supposée non vide (pas à vérifier) et qui renvoie la liste des couples `(a, q)` tels que `a` soit un élément de `l` et `q` soit la liste des autres éléments, dans l'ordre original. L'ordre des couples n'est pas imposé. Prouver la terminaison et la correction de la fonction et déterminer sa complexité.

Exemple : pour la liste `[2; 1; 4; 3]`, la fonction renverra la liste `([2, [1; 4; 3]); (1, [2; 4; 3]); (4, [2; 1; 3]); (3, [2; 1; 4])]` à l'ordre des couples près (on rappelle que les parenthèses n'étaient ici pas nécessaires par la bonne gestion des virgules et points-virgules).

Exercice $\beta 3$: Créer un type permettant de représenter une structure, **obligatoirement persistante pour espérer marquer des points sur cet exercice**, d'arbre binaire presque complet, en s'autorisant autant d'informations annexes que nécessaire pour maintenir l'invariant de structure. Écrire une fonction construisant à partir d'un tel arbre un arbre ayant un nœud en plus (on mettra la nouvelle étiquette en argument).

Deux suggestions (choisir l'une des deux ou en inventer une autre...) : mettre en plus de l'étiquette d'un nœud l'information de la taille du sous-arbre enraciné en ce nœud, ou la taille des sous-arbres gauche et droit.

Exercices en C

Exercice $\gamma 1$: Écrire une structure pour représenter un arbre binaire d'entiers. Avec cette structure, écrire une fonction qui détermine si un arbre binaire d'entiers est un ABR. Prouver la terminaison et la correction de la fonction et déterminer sa complexité.

Exercice $\gamma 2$: Écrire une fonction qui prend en argument un entier `n` et qui renvoie l'élément d'indice `n` de la suite de Conway en tant que chaîne de caractères.

Si la suite de Conway n'est pas connue, en voici la définition : à l'indice 0, il s'agit de 1 (en tant que nombre habituellement, mais ce sera une chaîne de chiffres pour nous vu la taille), puis pour tout entier n, à l'indice $n+1$ il s'agit de l'énonciation des chiffres de la suite à l'indice n, c'est-à-dire chaque chiffre de la suite précédé du nombre d'occurrences consécutives. Autrement dit, les valeurs sont 1 puis 11 (« un 1 »), 21 (« deux 1 »), 1211 (« un 2, un 1 »), 111221 (« un 1, un 2, deux 1 »)...

Exercice $\gamma 3$: On considère la structure de tas binaire presque complet d'entiers implémentée par un tableau redimensionnable d'entiers. Écrire une structure de tableau redimensionnable d'entiers puis écrire une fonction qui prend en argument une instance de cette structure, un indice `i` légitime dans le tableau correspondant et un entier `x` et qui mute l'élément à l'indice `i` pour le remplacer par `x` puis rétablit l'invariant de structure des tas par la suite. **Les tas seront des tas-min.**

Problème 1 : Listes doublement chaînées

On considère la structure de liste doublement chaînée, qui étend l'idée d'une liste chaînée en ajoutant pour chaque maillon un pointeur vers son prédécesseur (vide pour le premier élément) et la liste elle-même contient alors un deuxième pointeur vers son dernier élément. Voici une implémentation partielle possible en OCaml :

```
type 'a elem_ldc = { valeur : 'a; mutable suivant : 'a elem_ldc option;
    mutable precedent : 'a elem_ldc option };;

type 'a ldc = { mutable vide : bool; mutable debut : 'a elem_ldc option;
    mutable fin : 'a elem_ldc option };;

let creer_ldc () = { vide = true; debut = None; fin = None };;

let ajouter_debut_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if not l.vide then
  ( ebis.suivant <- l.debut;
    match l.debut with
    | Some el -> el.precedent <- Some ebis
    | _ -> failwith "Ce cas ne se produit pas" )
  else
  ( l.vide <- false; l.fin <- Some ebis );
  l.debut <- Some ebis;;

let ajouter_fin_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if not l.vide then
  ( ebis.precedent <- l.fin;
    match l.fin with
    | Some el -> el.suivant <- Some ebis
    | _ -> failwith "Ce cas ne se produit pas" )
  else
  ( l.vide <- false; l.debut <- Some ebis; );
  l.fin <- Some ebis;;

let retirer_debut_ldc l =
  match l.vide, l.debut with
  | true, _ -> failwith "Ldc vide"
  | false, None -> failwith "Cas impossible"
  | false, Some ebis -> let e = ebis.valeur in
      l.debut <- ebis.suivant;
      ( match l.debut with
       | None -> l.vide <- true; l.fin <- None;
       | Some deb -> deb.precedent <- None );
  e;;

let retirer_fin_ldc l =
  match l.vide, l.fin with
  | true, _ -> failwith "Ldc vide"
  | false, None -> failwith "Cas impossible"
  | false, Some ebis -> let e = ebis.valeur in
      l.fin <- ebis.precedent;
      ( match l.fin with
       | None -> l.vide <- true; l.debut <- None;
       | Some fi -> fi.suivant <- None );
  e;;
```

Question P1.1 : Écrire une implémentation possible pour des listes doublement chaînées d'entiers en C, que ce soit par « traduction » de ce qui précède ou non.

Question P1.2 : Écrire en OCaml une fonction d'ajout d'un maillon de valeur en argument après `k` maillons (aussi en argument). On considèrera que `k` est entre zéro et le nombre de maillons inclus.

Question P1.3 : Même exercice en ce qui concerne le retrait du maillon numéro `k`, avec cette fois-ci `k` ne pouvant pas être égal au nombre de maillons, et le premier maillon étant considéré comme d'indice `0`. On renverra la valeur retirée.

Désormais, nous allons faire deux tentatives d'accélération de l'opération précédente.

Dans un premier temps, le nombre de maillons sera ajouté comme information au niveau de la liste chaînée.

Question P1.4 : Préciser comment modifier la structure et les opérations élémentaires fournies ou écrites dans la première question du problème suivant le langage (qui est au choix). On pourra par exemple mentionner les lignes à modifier ou les endroits où ajouter des lignes qu'on écrira.

Question P1.5 : De même pour les questions P1.2 et P1.3. Si la question précédente a été faite en C, il faut alors écrire intégralement l'ajout et le retrait dans la nouvelle variante.

Question P1.6 : Calculer la complexité moyenne des deux fonctions écrites avant et après l'optimisation.

Deuxième tentative d'optimisation : on ajoute un pointeur vers le maillon du milieu (quand la taille est paire, c'est au choix mais il faut bien préciser ce que l'on a décidé et être cohérent dans la programmation).

Question P1.7 : Reprendre les questions P1.4 à P1.6 selon cette nouvelle contrainte, et en particulier comparer les complexités moyennes et commenter. Le langage peut être différent entre les questions reprises et celle-ci, mais ce n'est pas forcément la meilleure idée...

Problème 2 : Arbres AVL

Dans le cours, il était question d'arbres binaires dont chaque nœud avait deux sous-arbres de même hauteur à un près. En ajoutant la propriété d'ABR à de tels arbres, on tombe sur la sturcture d'arbre AVL (pas au programme), dont les opérations élémentaires d'ajout ou de retrait nécessitent également une certaine gymnastique.

Ici, nous n'allons pas nous intéresser aux opérations élémentaires mais plutôt poser d'autres questions sur ces arbres.

On rappelle que minimiser la taille d'un AVL à hauteur fixée revient, à partir de la hauteur 1, à utiliser d'un côté un AVL de hauteur un de moins et de l'autre un AVL de hauteur deux de moins, dans les deux cas minimisant eux-mêmes la taille d'un AVL de leur hauteur. La contrainte de cet énoncé est que le sous-arbre de hauteur un de moins que l'arbre soit le sous-arbre gauche (donc le sous-arbre droit est de hauteur deux de moins que l'arbre), et on appellera AVLF de tels arbres.

Dans un premier temps, on ne tiendra pas compte des étiquettes.

Question P2.1 : Construire les formes d'AVLF pour les hauteurs de 0 à 4, avec la convention que la hauteur d'un arbre vide soit -1 et la hauteur d'un arbre réduit à sa racine soit 0.

Question P2.2 : Donner en la prouvant la formule reliant la taille d'un AVLF à sa hauteur (à partir de la hauteur 0).

Question P2.3 : Pour la hauteur de 4 spécifiquement, montrer qu'un AVLF ne peut être associé à une coloration en faisant un arbre bicolore que d'une façon si on impose que la racine soit noire, et représenter cette coloration.

Question P2.4 : Dans le cadre de la question précédente, montrer que si la racine est mise en rouge depuis la coloration précédente on obtient une coloration correcte, et justifier par un argument très court qu'il n'en existe pas d'autre avec la racine en rouge.

Question P2.5 : Montrer que pour toute hauteur d'un AVLF il existe une coloration pour laquelle la racine est noire, et donner la méthode pour l'obtenir récursivement. Préciser alors la hauteur noire d'un AVLF vu comme un arbre bicolore.

À présent, pour les tailles qui ne correspondent pas à une taille d'AVLF, nous allons construire des arbres équilibrés tout de même.

Question P2.6 : Écrire dans un langage au choix une fonction qui prend en argument un entier `n` supposé strictement positif (pas à vérifier) et qui détermine l'entier `k` tel que `n` soit compris entre la taille d'un AVLF de hauteur `k` incluse et la taille d'un AVLF de hauteur un de plus que `k` exclue. Calculer la complexité de la fonction écrite et s'assurer qu'elle est au plus linéaire en la valeur de l'entrée.

Question P2.7 : Écrire dans un langage au choix une fonction qui prend en argument un entier `n` et qui renvoie une forme d'arbre binaire (structure à créer mais au choix) de sorte que sa hauteur soit la valeur renvoyée par la fonction précédente, le sous-arbre gauche soit de la même forme qu'un AVLF de même hauteur et le sous-arbre droit soit obtenu récursivement par la même fonction, avec comme cas de base les valeurs `0` et `1` pour `n` où il n'y a pas de choix possible sur la valeur de retour.

Désormais, les étiquettes seront incorporées.

Question P2.8 : Adapter la question précédente pour construire un AVL à partir d'une séquence (type au choix si c'est fait en OCaml) d'entiers, de sorte que la forme soit la même.

Question P2.9 : Calculer la complexité de la fonction précédente et commenter cette méthode d'obtention d'un ABR équilibré.

Annexe

On rappelle à toute fin utile des fonctions du module `Hashtbl` avec des informations sur leur spécification :

- `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
- `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
- `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
- `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
- `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
- `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.
- `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option pour éviter de lever une exception si la clé est absente ;
- `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage).

Pour la manipulation des chaînes en C, on propose les fonctions suivantes :

- `atoi(chaine)` renvoie l'entier représenté dans la chaîne en argument (ne pas l'utiliser si la chaîne ne correspond pas exactement à un entier), la fonction est `atof` pour renvoyer un flottant de type `double` ;
- `sprintf(chaine, "%d", n)` écrit dans la chaîne en premier argument l'entier en troisième argument, ce qui écrase la chaîne et suppose que sa taille soit suffisante, le format devient `"%f"` pour un flottant ;
- `strcat(chaine1, chaine2)` écrit à la suite de la chaîne en premier argument la chaîne en deuxième argument (même remarque sur la taille) ;
- `strcpy(chaine1, chaine2)` écrit au début de la chaîne en premier argument la chaîne en deuxième argument (idem).

Enfin, voici les règles du Bonus Club :

- La première règle du Bonus Club est : il est interdit de parler du Bonus Club.
- La deuxième règle du Bonus Club est : il est interdit de parler du Bonus Club (sérieusement, si j'entends quelqu'un commenter ceci en plein DS, j'annule le bonus).
- Troisième règle du Bonus Club : quelqu'un crie stop, quelqu'un s'écroule ou n'en peut plus, le DS est terminé (en tout cas pour cette personne).
- Quatrième règle : seulement une personne par copie.
- Cinquième règle : si vous avez plus de deux tiers des points aux questions de cours et que vous réclamez un bonus, vous le recevrez.
- Sixième règle : pas d'antisèche dans la chemise, ni dans les chaussures.
- Septième règle : le DS continuera aussi longtemps que nécessaire (mais pas plus que le temps alloué).
- Et huitième et dernière règle : si c'est le premier samedi de printemps 2024, vous devez vous battre (contre le sujet de DS).