Correction du DS 4

Julien Reichert

La correction des questions de cours étant dans le cours, elle ne sera pas donnée ici...

Exercices

Exercice α1 (en OCaml) :

```
let somme_maximale_chemin_matrice mat =
  let n = Array.length mat in assert (n > 0);
  let m = Array.length mat.(0) in assert (m > 0);
  let valeurs = Array.make_matrix n m 0 in
  let directions = Array.make_matrix n m 0 in (* ligne inutile cas de base *)
  valeurs.(0).(0) <- mat.(0).(0);
  for i = 1 to n-1 do
    valeurs.(i).(0) <- valeurs.(i-1).(0) + mat.(i).(0);
    directions.(i).(0) <- -1 (* haut, et l. i. cas de base *)
  done;
  for j = 1 to m-1 do
    valeurs.(0).(j) <- valeurs.(0).(j-1) + mat.(0).(j);
    directions.(0).(j) <- 1 (* gauche, et l. i. cas de base *)
  done;
  for i = 1 to n-1 do
    for j = 1 to m-1 do
      directions.(i).(j) <- if valeurs.(i-1).(j) > valeurs.(i).(j-1) then -1 else 1; (* l. i. *)
      valeurs.(i).(j) <- mat.(i).(j) + max valeurs.(i-1).(j) valeurs.(i).(j-1)
    done
  done;
  valeurs.(n-1).(m-1);;

(* Pour passer du cas de base à la reconstruction, remplacer la dernière ligne par :
  let rec construit_chemin res i j = (* récursion terminale ou List.rev à la place *)
    if i = 0 && j = 0 then (0, 0)::res
    else if directions.(i).(j) = -1 then construit_chemin ((i, j)::res) (i-1) j
    else construit_chemin ((i, j)::res) i (j-1)
  in construit_chemin [] (n-1) (m-1)
*)
```

Il s'agit d'un cas d'application de la programmation dynamique : la somme maximale d'un chemin jusqu'à une certaine case est la somme jusqu'à la case en question si elle est sur la première ligne ou sur la première colonne, et c'est la valeur de la case plus la somme maximale d'un chemin jusqu'à la voisine du haut ou de gauche (la plus grande des deux valeurs) sinon.

Pour consommer moins d'espace, on peut faire le calcul colonne par colonne ou ligne par ligne (la plus petite taille en priorité) car celles qui précèdent n'ont plus d'utilité une fois la suivante calculée. Malheureusement, l'adaptation qui calcule le chemin devient alors bien plus compliquée sans utiliser d'espace linéaire en la taille totale, et on ne se posera pas la question de comment faire ici.

Ainsi, la fonction écrite ci-avant répond aux deux questions, la deuxième étant délimitée par des commentaires.

En termes de complexité, la double boucle ayant une complexité constante à l'intérieur ne pose pas de souci de calcul (on additionne deux boucles ayant une complexité constante à l'intérieur pour l'initialisation pour un total inférieur au reste, ainsi qu'une création de deux matrices ayant également une complexité linéaire à la taille de l'entrée).

Enfin, en cas de vélléités de programmer en C, il faudra faire bien attention à la manipulation de structures de dimension 2 ainsi qu'à la gestion de la mémoire.

Exercice α2 (en OCaml) :

Dans un arbre, un nœud ne peut être rencontré qu'une fois au niveau du branchement d'une récursion, et donc la programmation dynamique n'est pas nécessaire pour éviter de lancer plusieurs fois le même appel récursif. Il s'agit toutefois de ne pas refaire la somme à chaque branche car celles-ci vont avoir un début commun.

```
type arbre_entiers = Noeud of int * arbre_entiers list;;
(* pas d'arbre vide, pas de constructeur spécial par choix *)

let somme_maximale_branche arbre =
  let rec maximise somme a = match a with
  | Noeud(x, []) -> somme + x
  | Noeud(x, l) -> List.fold_left (fun accu enfant -> max accu (maximise (somme+x) enfant)) 0 l
  in maximise 0 arbre;;
```

La complexité se détermine de la sorte : pour chaque nœud sans enfant, c'est constant ; pour un nœud avec une liste d'enfants, on appelle la fonction pour chacun et on fait pour chacun une comparaison par la fonction `max` dans le `fold`. Le coût est à considérer comme intégré à la constante dans l'appel concernant chaque enfant, ce qui fait un coût amorti constant des appels récursifs pour chaque nœud, d'où un total linéaire en la taille de l'arbre.

Exercice β1 :

```
type 'a arbre = Vide | Noeud of 'a * 'a arbre list;;
type 'a arbre_binaire = V | N of 'a arbre_binaire * 'a * 'a arbre_binaire;;

let convertit_arbre_vers_arbre_binaire arbre =
  let rec aux fratrie arb =
  match arb, fratrie with
  | Vide, [] -> V
  | Noeud(x, []), [] -> N(V, x, V)
  | Noeud(x, a::q), [] -> N(aux q a, x, V)
  | Vide, a::q -> aux q a
  | Noeud(x, []), a::q -> N(V, x, aux q a)
  | Noeud(x, a::q), b::r -> N(aux q a, x, aux r b)
  in aux [] arbre;;

let rec convertit_arbre_binaire_vers_arbre arbre =
  let rec reconstruit a = match a with
  | V -> Vide
  | N(V, x, _) -> Noeud(x, [])
  | N(g, x, _) -> Noeud(x, fratrie [] g)
  and fratrie l a = match a with
  | V -> List.rev l
  | N(_, _, d) -> fratrie (reconstruit a :: l) d
  in reconstruit arbre;;
```

On rappelle le principe de la transformation : dans l'arbre binaire obtenu à partir d'un arbre d'arité quelconque, le premier fils devient le fils gauche et le frère suivant devient le fils droit.

Exercice β2 :

```
let toutes_extractions l =
  let rec aux debut fin = match fin with
  | [] -> []
  | a::q -> (a, (List.rev debut)@q)::aux (a::debut) q
  in aux [] l;;
```

On note t la taille de la liste en argument.

La présence de l'opérateur @ peut être perturbante, mais puisque la taille du résultat est quadratique en t, on ne peut tout de même pas faire mieux.

La récursion s'arrête car chaque appel récursif déclenche un appel sur la queue de son deuxième argument, et la complexité est linéaire en t sur chaque appel, sachant que le nombre d'appels est alors aussi t, d'où un coût global quadratique comme anticipé.

En ce qui concerne la correction, on va plutôt faire une récurrence sur le numéro d'un appel récursif : au i-ième appel (pour i entre 1 et la taille de la liste, en particulier le premier est effectivement associé à $i = 1$), la taille de `debut` est $i - 1$ et la taille de `fin` est $t - (i - 1)$. C'est vrai pour $i = 1$ et un appel en déclenche un autre avec un élément en plus dans `debut` et un en moins dans `fin`. Plus précisément, les éléments de `debut` sont les premiers éléments de la liste en argument mais à l'envers, et les éléments de `fin` sont les derniers éléments de la liste en argument. Ainsi, coller `debut` à l'envers sur la queue de `fin` correspond à tous les éléments de la liste sauf un, et dans l'ordre de départ, et l'élément absent est précisément associé à ce recollement dans un couple à chacun des appels récursifs, donc on dispose de toutes les possibilités d'avoir un élément associé au reste en se servant de ce qui a été énoncé précédemment.

Exercice β3 :

Il est certes vrai que chaque nœud a au moins un sous-arbre de taille une puissance de deux moins un dans un arbre binaire presque complet, et conjuguer le test de cette propriété avec la comparaison des tailles permet de décider où aller à chaque étape, mais cela donne une complexité quadratique en la hauteur alors qu'on peut avoir une complexité linéaire en déterminant une fois pour toutes juste à partir de l'information sur la racine où aller. On économiserait alors du temps car la taille n'aurait même pas besoin d'être mise à chaque nœud, mais on ne va pas mélanger un type somme avec un type enregistrement et la structure devait être persistante...

Le chemin d'ajout s'obtient en écrivant la taille de l'arbre plus un en binaire, et en retirant le `1` initial. Les `0` sont alors des indications de partir dans le sous-arbre gauche et les `1` des indications de partir dans le sous-arbre droit. La formule elle-même s'intuite en reprenant la représentation d'un arbre binaire presque complet dans un tableau.

Il faut penser à adapter dans l'arbre retourné les tailles sur tous les nœuds (même si seule la taille écrite au niveau du premier nœud est consultée en pratique).

```
type 'a abpc = V | N of 'a * int * 'a abpc * 'a abpc;;
(* Pour une fois le fils gauche est énoncé après les informations. *)

let chemin_ajout arbre =
  match arbre with
  | V -> []
  | N(_, n, _, _) ->
    let rec aux i = if i = 0 then []
    else (i mod 2) :: aux (i/2)
    in List.tl (List.rev (aux (n+1)));;
```

```
let ajoute_noeud x arbre =
  if arbre = V then N(x, 1, V, V)
  else
    let liste = chemin_ajout arbre in
    let rec suivre_chemin a l =
    match a, l with
    | V, [] -> N(x, 1, V, V)
    | V, _ | _, [] -> failwith "Cas impossible"
    | N(y, n, g, d), 0::q -> N(y, n+1, suivre_chemin g q, d)
    | N(y, n, g, d), _::q -> N(y, n+1, g, suivre_chemin d q)
    in suivre_chemin arbre liste;;
```

Exercice γ1 :

```
struct a_b_i { int etiquette; struct a_b_i* fg; struct a_b_i* fd; };
typedef struct a_b_i iab;

bool est_abr(iab a)
{
  bool est_abr_aux(bool minobidon, int mino, bool majobidon, int majo, iab a)
  {
    if (!minobidon && a.etiquette < mino || !majobidon && a.etiquette > majo) return false;
    if (a.fg != NULL && !est_abr_aux(minobidon, mino, false, a.etiquette, *a.fg)) return false;
    if (a.fd != NULL && !est_abr_aux(false, a.etiquette, majobidon, majo, *a.fd)) return false;
    return true;
  }
  return est_abr_aux(true, 0, true, 0, a);
}
```

La terminaison se prouve dans la mesure où la structure créée représente un ensemble inductif et que la récursion de la fonction ci-avant porte sur des appels qui s'imbriquent pour des éléments plus petits selon l'ordre structurel. En termes de complexité, le traitement de chaque nœud donne lieu à une complexité constante et à des appels récursifs dont le coût est comptabilisé au niveau du traitement des nœuds concernés, pour un total qui est alors linéaire en la taille de l'arbre en argument.

Concernant la correction, elle se prouve par induction structurelle : un arbre binaire est un ABR si, et seulement si, chaque nœud a une étiquette plus grande que tous les nœuds de son sous-arbre gauche (ce qui se retrouve dans la contrainte de majoration imposée avec l'étiquette du nœud parent dans tout le sous-arbre gauche, et que cette contrainte est de plus en plus stricte au fur et à mesure où on va vers la gauche puisque le nouveau majorant est plus petit que le précédent d'après ce qui précède), et analogue en ce qui concerne le sous-arbre droit.

Exercice γ2 :

On commence par calculer la taille de la réponse pour chaque itération (dont une place pour le zéro terminal).

```
int taille_conway(char* chaine)
{
  int reponse = 1;
  for (int i = 0 ; chaine[i] != '\0' ; i += 1)
  {
    if (chaine[i] != chaine[i+1]) // dont les cas où chaine[i+1] est '\0'
      reponse += 2;
  }
  return reponse;
}
```

```
char* conway(int n)
{
  char* reponse = malloc(2*sizeof(char));
  reponse[0] = '1';
  reponse[1] = '\0';
  for (int i = 1 ; i <= n ; i += 1)
  {
    int taille = taille_conway(reponse);
    char* nouveau = malloc(taille * sizeof(char));
    int consec = 1;
    int indice_nouveau = 0;
    for (int j = 0 ; reponse[j] != '\0' ; j += 1)
    {
      if (reponse[j] == reponse[j+1]) consec += 1;
      else
      {
        nouveau[indice_nouveau] = (char) (consec + 48); // 48 étant (int) '0'
        nouveau[indice_nouveau+1] = reponse[j];
        consec = 1;
        indice_nouveau += 2;
      }
    }
    nouveau[indice_nouveau] = '\0';
    free(reponse);
    reponse = nouveau; // Manipulation de pointeur toute simple
  }
  return reponse;
}
```

Exercice γ3 :

On reprend la correction du TP 7 :

```
struct t_r_i { int taille; int capacite; int* donnees; };

typedef struct t_r_i itr;

itr creer_itr(int capacite)
{
  itr t = { .taille = 0, .capacite = capacite, .donnees = malloc(capacite * sizeof(int)) };
  return t;
}

int acces_itr(itr t, int i)
{
  assert(0 <= i && i < t.taille);
  return t.donnees[i];
}

void modif_itr(itr t, int i, int x)
{
  assert(0 <= i && i < t.taille);
  t.donnees[i] = x;
}
```

```
void redimensionne_itr(itr* t)
{
  int nv_capacite = t->capacite * 2;
  t->capacite = nv_capacite;
  int* nv_tab = malloc(nv_capacite * sizeof(int));
  for (int i = 0 ; i < t->taille ; i += 1)
  {
    nv_tab[i] = t->donnees[i];
  }
  free(t->donnees);
  t->donnees = nv_tab;
}

void append_itr(itr* t, int x)
{
  if (t->taille == t->capacite) redimensionne_itr(t);
  t->donnees[t->taille] = x;
  t->taille += 1;
}

int pop_itr(itr* t)
{
  assert(t->taille > 0);
  int rep = t->donnees[t->taille-1];
  t->taille -= 1;
  return rep;
}

void mute_tas(itr t, int i, int x) // Pas besoin de pointeur pour t, seul le tableau est muté
{
  void corrige_bas(int i)
  {
    if (2 * i + 2 < t.taille && acces_itr(t, 2*i+2) < acces_itr(t, 2*i+1) && acces_itr(t, 2*i+2) < x)
    {
      modif_itr(t, i, acces_itr(t, 2*i+2));
      modif_itr(t, 2*i+2, x);
      corrige_bas(2*i+2);
    }
    else if (2 * i + 1 < t.taille && acces_itr(t, 2*i+1) < x)
    {
      modif_itr(t, i, acces_itr(t, 2*i+1));
      modif_itr(t, 2*i+1, x);
      corrige_bas(2*i+1);
    }
  }
  void corrige_haut(int i)
  {
    if (i > 0 && acces_itr(t, (i-1)/2) > x)
    {
      modif_itr(t, i, acces_itr(t, (i-1)/2));
      modif_itr(t, (i-1)/2, x);
      corrige_haut((i-1)/2);
    }
  }
  modif_itr(t, i, x); corrige_bas(i); corrige_haut(i); // Un des deux au moins s'arrête tout de suite.
}
```

Problème 1

Question P1.1 :

On peut aussi reprendre la correction du TP 7 et adapter...

Le booléen indiquant que la liste chaînée est vide n'est en pratique pas nécessaire, et on ne s'en servira pas.

Quelques lignes cruciales ont été signalées par un commentaire. L'implémentation fournie en OCaml aura éventuellement permis d'y penser.

```
struct m_l_d_c_i { int valeur; struct m_l_d_c_i* suivant;
struct m_l_d_c_i* precedent; };
typedef struct m_l_d_c_i mildc;

struct l_d_c_i { mildc* debut; mildc* fin; };
typedef struct l_d_c_i ildc;

ildc creer_ildc()
{
  ildc l = { .debut = NULL, .fin = NULL };
  return l;
}

void ajouter_debut_ildc(ildc* l, int x)
{
  mildc* xm = malloc(sizeof(mildc));
  xm->valeur = x;
  xm->suivant = l->debut;
  xm->precedent = NULL;
  if (l->debut != NULL) l->debut->precedent = xm; // Ne pas oublier !
  else l->fin = xm; // Ne surtout pas oublier !
  l->debut = xm;
}

void ajouter_fin_ildc(ildc* l, int x) // Symétrique avec fin
{
  mildc* xm = malloc(sizeof(mildc));
  xm->valeur = x;
  xm->precedent = l->fin;
  xm->suivant = NULL;
  if (l->fin != NULL) l->fin->suivant = xm;
  else l->debut = xm;
  l->fin = xm;
}

int retirer_debut_ildc(ildc* l)
{
  assert (l->debut != NULL);
  mildc* p = l->debut;
  int rep = p->valeur;
  l->debut = p->suivant;
  if (l->debut == NULL) l->fin = NULL; // Sans le test, risque de segfault
  else l->debut->precedent = NULL; // Ne pas oublier !
  free(p);
  return rep;
}
```

```
int retirer_fin_ildc(ildc* l)
{
  assert (l->fin != NULL);
  mildc* p = l->fin;
  int rep = p->valeur;
  l->fin = p->precedent;
  if (l->fin == NULL) l->debut = NULL;
  else l->fin->suivant = NULL;
  free(p);
  return rep;
}
```

Question P1.2 :

Pour éviter d'avoir un filtrage pour savoir si on doit modifier l'information du début, le cas du premier indice sera géré par un test initial. Par contre, le changement à la fin ne pourra pas être anticipé sans récursion.

Une version antérieure du corrigé utilisait des maillons et non des options de maillons comme arguments dans la récursion, la gestion de l'ajout avant le dernier était pénible...

```
let ajouter_avant_ldc l e k =
  if k = 0 then ajouter_debut_ldc l e else
  let rec aux maillonoption i = match maillonoption with
  | None when i < k -> failwith "Dépassement d'indice"
  | None ->
    let ebis = { valeur = e; suivant = None; precedent = l.fin } in
    ( match l.fin with
    | None -> failwith "Cas déjà traité"
    | Some mbis -> mbis.suivant <- Some ebis; l.fin <- Some ebis )
  | Some { suivant = mop } when i < k -> aux mop (i+1)
  | Some m ->
    let ebis = { valeur = e; suivant = maillonoption; precedent = None } in
    ( match m.precedent with
    | None -> failwith "Cas déjà traité"
    | Some mbis ->
      m.precedent <- Some ebis;
      mbis.suivant <- Some ebis;
      ebis.precedent <- Some mbis )
  in aux l.debut 0;;
```

Question P1.3 :

```
let retirer_indice_ldc l k =
  if k = 0 then retirer_debut_ldc l else
  let rec aux maillonoption i = match maillonoption with
  | None -> failwith "Dépassement d'indice"
  | Some { valeur = e; suivant = mop2; precedent = Some m1 } when i = k ->
    m1.suivant <- mop2;
    ( match mop2 with
      | Some m2 -> m2.precedent <- Some m1
      | None -> l.fin <- Some m1 ); (* On a retiré le dernier, et pas le premier sinon cas déjà traité *)
    e (* ne pas oublier ! *)
  | Some { suivant = mop } -> aux mop (i+1)
  in aux l.debut 0;;
```

Question P1.4 :

La question suivante suggère de faire le travail en OCaml.

Le booléen précisant la vacuité n'a plus d'intérêt car il sera redondant avec la taille.

Le type pour les maillons reste le même, et le type des listes est modifié (on va garder le nom pour raccourcir).

```
type 'a ldc = { mutable taille : int; mutable debut : 'a elem_ldc option;
    mutable fin : 'a elem_ldc option };;

let creer_ldc () = { taille = 0 ; debut = None; fin = None };; (* changement *)

let ajouter_debut_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if l.taille > 0 then (* changement *)
  ( ebis.suivant <- l.debut;
    match l.debut with
    | Some el -> el.precedent <- Some ebis
    | _ -> failwith "Ce cas ne se produit pas" )
  else l.fin <- Some ebis; (* changement *)
  l.debut <- Some ebis;
  l.taille <- l.taille + 1;; (* changement *)

let ajouter_fin_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if l.taille > 0 then (* changement *)
  ( ebis.precedent <- l.fin;
    match l.fin with
    | Some el -> el.suivant <- Some ebis
    | _ -> failwith "Ce cas ne se produit pas" )
  else l.debut <- Some ebis; (* changement *)
  l.fin <- Some ebis;
  l.taille <- l.taille + 1;; (* changement *)

let retirer_debut_ldc l =
  match l.taille, l.debut with (* changement *)
  | 0, _ -> failwith "Ldc vide" (* changement *)
  | _, None -> failwith "Cas impossible" (* changement *)
  | _, Some ebis -> l.taille <- l.taille - 1; (* changement *)
  let e = ebis.valeur in l.debut <- ebis.suivant;
      ( match l.debut with
        | None -> l.fin <- None; (* changement *)
        | Some deb -> deb.precedent <- None );
  e;;

let retirer_fin_ldc l =
  match l.taille, l.fin with (* changement *)
  | 0, _ -> failwith "Ldc vide" (* changement *)
  | _, None -> failwith "Cas impossible" (* changement *)
  | _, Some ebis -> l.taille <- l.taille - 1; (* changement *)
  let e = ebis.valeur in l.fin <- ebis.precedent;
      ( match l.fin with
        | None -> l.debut <- None; (* changement *)
        | Some fi -> fi.suivant <- None );
  e;;
```

Question P1.5 :

Ce n'est pas du copier-coller mais une adaptation en direct d'un code déjà existant, qui est à considérer comme effacé (mais l'effacer n'arrangerait personne, n'est-ce pas ?)...

```
let ajouter_avant_ldc l e k =
  if k < 0 || k > l.taille then failwith "Dépassement d'indice";
  if k = 0 then ajouter_debut_ldc l e else
  if k = l.taille then ajouter_fin_ldc l e else
  let rec aux mop i = match mop with (* moins de cas grâce à l'information de la taille *)
  | Some { suivant = mop2 } when i < k -> aux mop2 (i+1)
  | Some { precedent = mop2 } when i > k -> aux mop2 (i-1)
  | Some m ->
    let ebis = { valeur = e; suivant = mop; precedent = None } in
    ( match m.precedent with
     | None -> failwith "Cas déjà traité"
     | Some mbis ->
    m.precedent <- Some ebis;
    mbis.suivant <- Some ebis;
    ebis.precedent <- Some mbis )
  | _ -> failwith "Cas déjà traité"
  in if k < l.taille / 2 then aux l.debut 0 else aux l.fin (l.taille-1);
  l.taille <- l.taille + 1;; (* ne pas oublier ! *)

let retirer_indice_ldc l k =
  if k < 0 || k >= l.taille then failwith "Dépassement d'indice";
  if k = 0 then retirer_debut_ldc l else
  if k = l.taille-1 then retirer_fin_ldc l else
  let rec aux mop i = match mop with
  | None -> failwith "Cas impossible"
  | Some { valeur = e; precedent = Some m1; suivant = mop2 } when i = k ->
    m1.suivant <- mop2;
    ( match mop2 with
      | Some m2 -> m2.precedent <- Some m1
      | _ -> failwith "Cas déjà traité" );
    l.taille <- l.taille - 1; e (* faire ici la mutation *)
  | Some { suivant = mop } when i < k -> aux mop (i+1)
  | Some { precedent = mop } -> aux mop (i-1)
  in if k < l.taille / 2
  then aux l.debut 0
  else aux l.fin (l.taille - 1);;
```

Question P1.6 :

On va choisir le nombre d'appels récursifs à la fonction auxiliaire comme unité de complexité. L'ajout de l'information de la taille n'alourdit pas vraiment le travail avant la récursion ni pendant.

Sans optimisation : en moyenne $\frac{n}{2}$ (plus une valeur constante donc négligeable) appels récursifs où n est la taille de la liste chaînée.

Avec optimisation : en moyenne $\frac{n}{4}$ (plus une valeur constante donc négligeable) appels récursifs car c'est la moyenne commune du nombre d'appels récursifs dans chacun des scénarions.

L'optimisation est donc fortement recommandée, d'autant plus qu'elle ne rend pas la structure plus compliquée à gérer en termes de programmation et que le coût en espace pour passer d'un booléen à un entier reste raisonnable.

Question P1.7 :

On prendra pour élément médian celui de gauche si la taille est paire.

Modulariser est la clé, vu que les six opérations élémentaires de modification feront en fonction de la parité un décalage du pointeur vers le milieu.

```
type 'a ldc = { mutable taille : int; mutable debut : 'a elem_ldc option;
    mutable fin : 'a elem_ldc option; mutable milieu : 'a elem_ldc option };;

let decaler_milieu_ldc l avancer = (* avancer est un booléen indiquant s'il faut avancer le milieu *)
  match l.milieu, avancer with
  | Some { suivant = mop }, true -> l.milieu <- mop
  | Some { precedent = mop }, false -> l.milieu <- mop
  | _ -> failwith "La fonction n'aurait pas été appelée dans ce cas";;

let creer_ldc () = { taille = 0 ; debut = None; fin = None; milieu = None };;

let ajouter_debut_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if l.taille > 0 then
  ( ebis.suivant <- l.debut;
    ( match l.debut with
     | Some el -> el.precedent <- Some ebis
     | _ -> failwith "Ce cas ne se produit pas" );
    if l.taille mod 2 = 1 then decaler_milieu_ldc l false
  )
  else ( l.fin <- Some ebis; l.milieu <- Some ebis );
  l.debut <- Some ebis;
  l.taille <- l.taille + 1;;

let ajouter_fin_ldc l e =
  let ebis = { valeur = e; suivant = None; precedent = None } in
  if l.taille > 0 then
  ( ebis.precedent <- l.fin;
    ( match l.fin with
      | Some el -> el.suivant <- Some ebis
      | _ -> failwith "Ce cas ne se produit pas" );
    if l.taille mod 2 = 0 then decaler_milieu_ldc l true
  )
  else ( l.debut <- Some ebis; l.milieu <- Some ebis );
  l.fin <- Some ebis;
  l.taille <- l.taille + 1;;

let retirer_debut_ldc l =
  match l.taille, l.debut with
  | 0, _ -> failwith "Ldc vide"
  | _, None -> failwith "Cas impossible"
  | _, Some ebis -> l.taille <- l.taille - 1;
  let e = ebis.valeur in l.debut <- ebis.suivant;
      ( match l.debut with
      | None -> l.fin <- None; l.milieu <- None
      | Some deb -> deb.precedent <- None );
      if l.taille mod 2 = 1 then decaler_milieu_ldc l true;
  e;;
```

```
let retirer_fin_ldc l =
  match l.taille, l.fin with
  | 0, _ -> failwith "Ldc vide"
  | _, None -> failwith "Cas impossible"
  | _, Some ebis -> l.taille <- l.taille - 1;
  let e = ebis.valeur in l.fin <- ebis.precedent;
      (match l.fin with
      | None -> l.debut <- None; l.milieu <- None
      | Some fi -> fi.suivant <- None);
      if l.taille > 0 && l.taille mod 2 = 0 then decaler_milieu_ldc l false;
  e;;

let ajouter_avant_ldc l e k =
  if k < 0 || k > l.taille then failwith "Dépassement d'indice";
  if k = 0 then ajouter_debut_ldc l e else
  if k = l.taille then ajouter_fin_ldc l e else
  let rec aux mop i = match mop with
  | Some { suivant = mop2 } when i < k -> aux mop2 (i+1)
  | Some { precedent = mop2 } when i > k -> aux mop2 (i-1)
  | Some m ->
    let ebis = { valeur = e; suivant = mop; precedent = None } in
    ( match m.precedent with
     | None -> failwith "Cas déjà traité"
     | Some mbis -> m.precedent <- Some ebis; mbis.suivant <- Some ebis; ebis.precedent <- Some mbis;
    )
  | _ -> failwith "Cas déjà traité"
  in if k < l.taille / 4 then aux l.debut 0
  else if k > (3 * l.taille) / 4 then aux l.fin (l.taille-1)
  else aux l.milieu ((l.taille-1) / 2);
  if (l.taille mod 2 = 1 && k <= l.taille / 2) then decaler_milieu_ldc l false;
  if (l.taille mod 2 = 0 && k >= l.taille / 2) then decaler_milieu_ldc l true;
  l.taille <- l.taille + 1;;

let retirer_indice_ldc l k =
  if k < 0 || k >= l.taille then failwith "Dépassement d'indice";
  if k = 0 then retirer_debut_ldc l else
  if k = l.taille-1 then retirer_fin_ldc l else
  let rec aux mop i = match mop with
  | None -> failwith "Cas impossible"
  | Some { valeur = e; precedent = Some m1; suivant = mop2 } when i = k ->
    m1.suivant <- mop2;
    ( match mop2 with
      | Some m2 -> m2.precedent <- Some m1
      | _ -> failwith "Cas déjà traité" );
    if l.taille mod 2 = 1 && k >= l.taille / 2 then decaler_milieu_ldc l false;
    if l.taille mod 2 = 0 && k < l.taille / 2 then decaler_milieu_ldc l true;
    l.taille <- l.taille - 1; e
  | Some { suivant = mop } when i < k -> aux mop (i+1)
  | Some { precedent = mop } -> aux mop (i-1)
  in if k < l.taille / 4 then aux l.debut 0
  else if k > (3 * l.taille) / 4 then aux l.fin (l.taille - 1)
  else aux l.milieu ((l.taille-1) / 2);;
```

Comme on pouvait s'y attendre, cette fois-ci le nombre maximal d'appels récursifs sera le quart de la taille donc le nombre moyen en sera le huitième, mais au prix de complications sévères dans l'implémentation. C'est donc fait mais pas à refaire, et avec quelques mois de retard Anas aura donc la réponse à sa question !

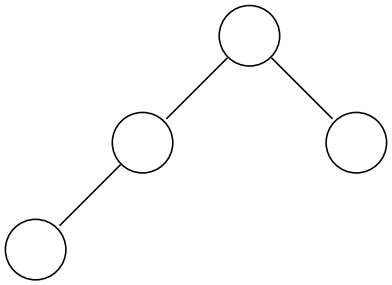
Problème 2

Question P2.1 :

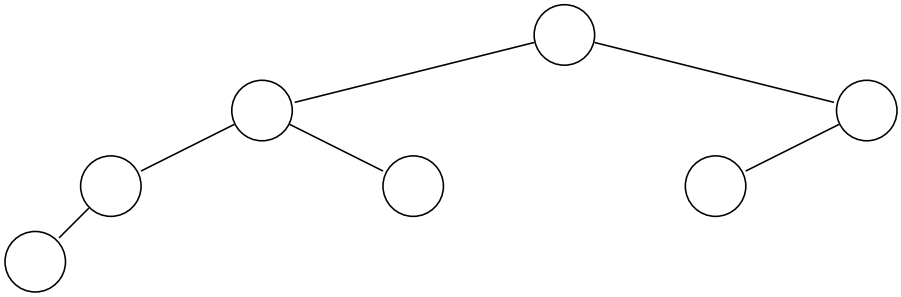
Hauteur 0 : un seul nœud.

Hauteur 1 : un nœud avec un fils gauche sans enfant et un fils droit vide.

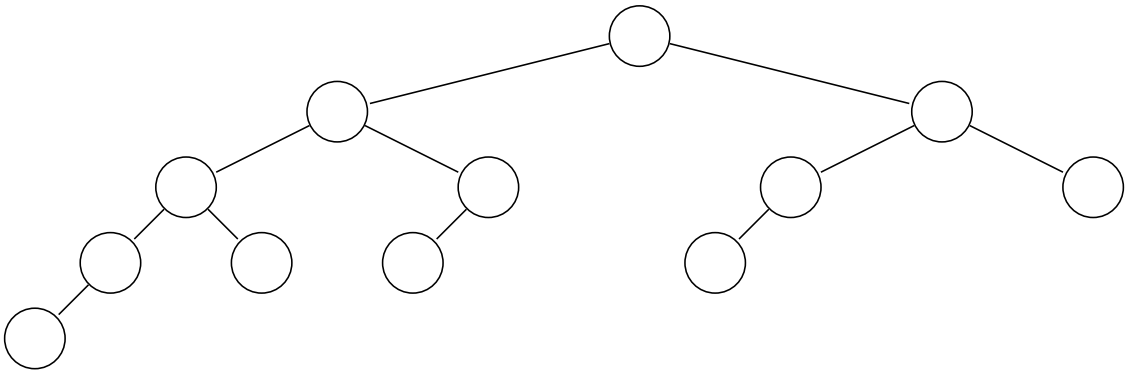
Hauteur 2 :



Hauteur 3 :



Hauteur 4 :



Question P2.2 :

On rappelle la formule obtenue dans la correction de l'exercice 7 du TD 11 : soit t_n la taille d'un AVLF de hauteur n, alors $t_0 = 1$, $t_1 = 2$ et pour tout entier naturel n, $t_{n+2} = t_{n+1} + t_n + 1$, d'où $t_n + 1$ est le $n+3$-ième terme de la suite de Fibonacci.

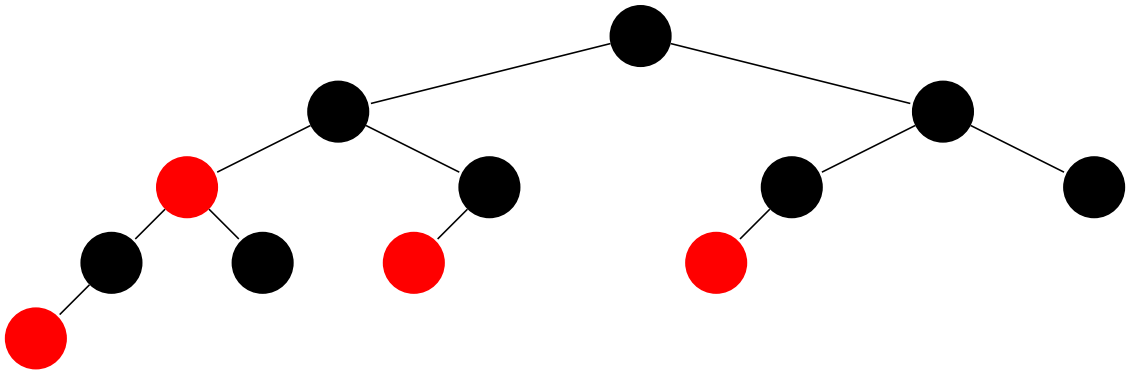
En calculant le terme général de la suite et en décalant de trois crans puis en retranchant un, on obtient

$$t_n = \frac{(\frac{1+\sqrt{5}}{2})^{n+3} - (\frac{1-\sqrt{5}}{2})^{n+3}}{\sqrt{5}} - 1.$$

Question P2.3 :

Puisque la hauteur est de 4, en admettant que la racine est noire, la hauteur noire doit être d'au moins 3 en raison de la présence d'une branche de taille 5 qui ne peut en particulier contenir qu'au plus deux nœuds rouges. Cependant, il existe aussi une branche de taille 3, donc la hauteur noire est également au plus de 3, d'où la valeur exacte et l'obligation de mettre en noir le fils droit de la racine et les deux enfants de celui-ci (dont le fils gauche car il est à la fin d'une branche par son absence de fils droit), laissant en rouge le dernier nœud du sous-arbre droit de la racine.

De même, il existe une branche de taille trois qui part de la racine et va à gauche puis à droite et s'arrête faute de fils droit, donc le fils gauche de la racine et le fils droit de celui-ci sont en noir, ce qui laisse le fils gauche de ce dernier en rouge. Pour ne pas avoir deux nœuds rouges consécutifs, dans la branche partant tout à gauche, il faut désormais alterner les couleurs, et le dernier nœud a également sa couleur imposée car son père est rouge. On a donc trouvé une coloration unique.



Question P2.4 :

Puisque les deux fils de la racine sont noirs, la racine aurait pu être rouge sans contredire les deux règles principales des arbres bicolores. Ceci étant, la couleur de la racine ne change que la hauteur noire de l'arbre et toutes les autres contraintes de la question précédente demeurent, d'où l'unicité de la coloration dans ce cas aussi. (L'argument le plus simple est que s'il y avait deux colorations avec la racine en rouge, noircir la racine donnerait deux colorations avec la racine en noir, ce qu'on a déjà exclu.)

Question P2.5 :

En pratique, le résultat est plus fort : la coloration est unique à la couleur de la racine près, qui est par ailleurs au choix si, et seulement si, la hauteur est paire.

Nous allons montrer un résultat parallèle :

Pour toute hauteur h d'un AVLF, il existe une coloration de l'arbre où la racine est noire et, si la hauteur est paire et non nulle, les deux fils de la racine sont noirs, cette coloration donnant lieu à une hauteur noire de $\lfloor \frac{h}{2} \rfloor + 1$.

La preuve se fait par récurrence double (on pouvait s'y attendre...) : on initialise pour $h = 0$ avec une racine noire et pour $h = 1$ avec une racine noire et un enfant rouge.

Par la suite, pour passer de la véracité de la propriété pour deux hauteurs consécutives, on fait une distinction de cas en fonction de la parité, en rappelant qu'un AVLF de hauteur $h + 2$ a un AVLF de hauteur $h + 1$ comme sous-arbre gauche et un AVLF de hauteur h comme sous-arbre droit.

Si la hauteur est paire (notée $2m+2$) : par hypothèse de récurrence double, on considère l'AVLF de hauteur $2m+1$ qu'on a colorié avec une hauteur noire de $m + 1$ et l'AVLF de hauteur $2m$ qu'on a aussi colorié avec une hauteur noire de $m + 1$. Alors en considérant une racine noire et ces deux AVLF comme fils, on obtient une coloration correcte avec une hauteur noire de $m + 2$, qui est bien $\lfloor \frac{2m+2}{2} \rfloor + 1$ et les deux fils de la racine sont effectivement noirs.

Si la hauteur est impaire (notée $2m+3$) : par hypothèse de récurrence double, on considère l'AVLF de hauteur $2m+2$ qu'on a colorié avec une hauteur noire de $m+2$ **mais en mettant ensuite la racine en rouge** (l'hypothèse nous l'autorise car les fils de la racine sont supposés noirs), **baissant d'un la hauteur noire** et l'AVLF de hauteur $2m+1$ qu'on a colorié avec une hauteur noire de $m+1$. Alors en considérant une racine noire et ces deux AVLF comme fils, on obtient une coloration correcte avec une hauteur noire de $m+2$, qui est bien $\lfloor \frac{2m+3}{2} \rfloor + 1$.

La propriété est donc héréditaire et vraie pour toute hauteur.

Question P2.6 :

La récursion peut effrayer en OCaml, autant partir sur C pour faire des boucles.

```
int pgifmui(int n) // plus grand indice Fibonacci moins un inférieur
{
  assert (n > 0);
  if (n == 1) return 0;
  if (n == 2) return 1;
  int k = 1;
  int nk = 2;
  int nkmu = 1;
  while (nk <= n)
  {
    int buff = nk;
    nk = nk + nkmu + 1;
    nkmu = buff; // On peut aussi faire nk - nkmu - 1.
    k += 1;
  }
  return k-1;
}
```

La complexité est logarithmique en la valeur de n puisque le nombre de tours de boucle est la réponse avec un traitement constant avant et dans la boucle. Le lien entre réponse et argument permet de conclure.

Question P2.7 :

Bon, en pratique la manipulation d'arbres suggère OCaml, donc on réécrit la fonction précédente (grâce à la boucle, c'est plus facile, en pratique)...

On lance un petit coup de mémoïzation dans une version adaptée de `pgifmui` pour muter un tableau et ne calculer la taille d'un AVLF à hauteur fixée qu'une fois.

```
type forme_avl = Videf | Noeudf of forme_avl * forme_avl;;

let pgifmui n =
  if n <= 0 then failwith "On voulait n strictement positif";
  let rec aux nk nkmu k =
    if nk > n then k-1
    else aux (nk + nkmu + 1) nk (k+1)
  in aux 2 1 1;;

let fmui k =
  let rep = Array.make (k+1) 2 in (* Astuce si k = 0 *)
    rep.(0) <- 1;
    for i = 2 to k do
      rep.(i) <- rep.(i-2) + rep.(i-1) + 1
    done; rep;;
```

```
let construit_forme_avl taille =
  if taille = 0 then Videf
  else let hauteur = pgifmui taille in
    let tailles = fmui hauteur in
    let rec construction n k =
      if n = 0 then Videf
      else if n = 1 then Noeudf(Videf, Videf)
      else
        let tailleg = tailles.(k-1) in
        let tailled = n - tailleg - 1 in
        let kd = if tailled >= tailles.(k-1) then k-1 else k-2 in
        Noeudf(construction tailleg (k-1), construction tailled kd)
    in construction taille hauteur;;
```

De manière amusante, la taille du sous-arbre droit de la racine peut devenir supérieure à la taille du sous-arbre gauche quand on approche du moment où la hauteur doit augmenter.

Question P2.8 :

Méthode : on trie la séquence (idéalement un tableau au vu de ce qui suit), puis on récurse sur les tranches où on récupère les valeurs, et on les place quand les tranches sont de taille un. Le tri lui-même ne sera pas le cœur du travail donc on autorise les raccourcis (à condition de les connaître !) sans pénalité.

```
type 'a avl = Vide | Noeud of 'a avl * 'a * 'a avl;;

let construit_avl tab =
  let taille = Array.length tab in
  if taille = 0 then Vide
  else let hauteur = pgifmui taille in
    let tabbis = Array.copy tab in
    let tailles = fmui hauteur in
    Array.sort compare tabbis;
    let rec construction debut fin k =
      let n = fin - debut + 1 in
      if n = 0 then Vide
      else if n = 1 then Noeud(Vide, tabbis.(debut), Vide)
      else
        let tailleg = tailles.(k-1) in
        let tailled = n - tailleg - 1 in
        let kd = if tailled >= tailles.(k-1) then k-1 else k-2 in
        let sag = construction debut (debut + tailleg - 1) (k-1) in
        let sad = construction (debut + tailleg + 1) fin kd in
        Noeud(sag, tabbis.(debut + tailleg), sad)
    in construction 0 (taille-1) hauteur;;
```

Question P2.9 :

Après avoir sélectionné un tri de complexité optimale, le reste est en temps linéaire en la taille de l'arbre formé. En pratique l'utilisation d'arbres bicolores permet d'éviter le tri pour une même complexité asymptotique, le mélange (petite sécurité) puis l'insertion dans un simple ABR aura la même complexité en moyenne et sera bien plus facile, et quitte à trier autant utiliser des arbres binaires presque complets, on échappe à toutes ces considérations sur les AVL et sur les nombres de Fibonacci (mais attention aussi à la disposition dans le tableau support de l'arbre binaire presque complet, ce tableau ne sera pas lui-même trié, on pourra par exemple le remplir en simulant un parcours en profondeur).