Projet d'Algorithmie (Rapport d'expérimentations sur les ABR et les AVL)

Auteur(s) : Florian Legendre

Légendes et Abbréviations utilisées

```
1 Ceci est du code source.
2 Selon les langages, différents mots seront colorés selon
3 si ce sont des mots clefs ou non (comme int, char, etc.).
```

Listing 1 – Exemple de code source

```
Ceci est un formattage automatique Latex d'un texte copié-collé
directement depuis un terminal Bash ayant valeur de capture
d'écran. La coloration correspond à une coloration quelconque
d'un terminal Bash (les chemins étant habituellement coloré et
le nom de l'utilisateur aussi comme crex@crex:~$ ...)
```

Listing 2 – Exemple d'une pseudo capture d'écran d'un terminal de commande (type eshell ou Bash)

Table des matières

Première partie

Exercice 1 - Arbres binaires de recherche

Première question : générer des ABR aléatoirement

Question : À partir du module sur les ABR que vous avez réalisé au TP4 et en utilisant la fonction Random.self_init du module Random qui permet d'initialiser un générateur de nombres aléatoires et la fonction Random.int borne qui donne un nombre aléatoire compris entre 0 et borne - 1 (le paramètre borne doit être inférieur à 2^{30}), écrivez une fonction bst_rnd_create qui crée un arbre binaire de recherche.

Réponse :
Pour la génération aléatoire de nos ABR nous avons opté pour une fonction itérative et un ABR mutable. Nous incrémentons cet arbre jusqu'à atteindre la taille désirée grâce à une fonction de notre cru du module BST :

```
1  let bst_rnd_create(bound, treeSize : int * int) : int bst =
2
3    Random.self_init();
4
5    let empty_tree : int bst = empty() in
6    let randABR : int bst ref = ref empty_tree in
7
8    for i=1 to treeSize do
9      let randInt : int = Random.int bound in
10     randABR := bst_linsert(!randABR, randInt);
11   done;
12
13   !randABR
14 ;;
```

Listing 3 – Code source de génération aléatoire d'arbres binaires de recherche

Nous obtenons alors, par exemple, ce genre d'arbres ABR :

```
# show_int_btree(bst_rnd_create(100, 10));;
    ___8____
   /        \
  1          _81_
            /    \
          17_     97
         /   \
       16     81
      /      /
     14     35
           /
          29
# show_int_btree(bst_rnd_create(100, 10));;
       __38__
      /      \
     11       61_
    /  \     /   \
   6    32  45    66
  /             /  \
 1             62   72
```

Listing 4 – Exemple d'arbres produits aléatoirement par ce code

---

Deuxième question : Calculer la moyenne des déséquilibres

Question : Le déséquilibre d'un arbre est la différence entre la hauteur de son fils gauche et la hauteur de son fils droit. À l'aide de la fonction bst_rnd_create estimez le déséquilibre moyen des arbres binaires de recherche construits à partir de suites de nombres entiers aléatoires. Pour avoir des estimations significatives, il est nécessaire de répéter un grand nombre de fois l'estimation de la moyenne faite sur un grand nombre d'arbres.

Réponse :

En suivant la définition fournie par l'énoncé et les axiomes du cours nous avons donc défini la fonction déséquilibre :

```
let desequilibre(tree : 'a t_btree) : int =
  if(isEmpty(tree))
  then 0

  else
    height(lson(tree)) - height(rson(tree))
;;
```

Listing 5 – Code source pour le calcul du déséquilibre

Nous avons ensuite utilisé cette fonction pour réaliser une fonction de calcul de la moyenne du déséquilibre des arbres ABR puis du calcul de la moyenne de ces moyennes :

```
let avgDesequilibre(sampleSize, treeSize : int * int) : float =

  let sum : float ref = ref 0. in

  for i=1 to sampleSize do
    sum := !sum +. float_of_int(desequilibre(bst_rnd_create(100, treeSize))
    )
  done;

  !sum /. float_of_int(sampleSize)
;;


let avgAvgDesequilibre(nbAvgs, sampleSize, treeSize : int * int * int) :
    float =

  let sum : float ref = ref 0. in

  for i=1 to nbAvgs do
    sum := !sum +. avgDesequilibre(sampleSize, treeSize);
  done;

  !sum /. float_of_int(nbAvgs)
;;
```

Listing 6 – Code source pour le calcul de la moyenne des moyennes des déséquilibres

Ce qui donne lieu aux résultats expérimentaux ci-dessous :

```
# avgAvgDesequilibre(100, 100, 100);;
- : float = 0.190800000000000053
# avgAvgDesequilibre(100, 100, 100);;
- : float = 0.0587999999999999357
# avgAvgDesequilibre(100, 100, 100);;
- : float = 0.178500000000000048
```

Listing 7 – Résultat de l'exécution répétée du calcul de la moyenne des moyennes de déséquilibres

Conclusion : On constate alors que ces moyennes sont autour de 0. Ce qui signifie que lorsque les valeurs à insérer sont très diversifiées les arbres binaires de recherche peuvent être des structures de données tout à fait adaptées. Ce qui risque de ne pas être le cas lorsque les données ne sont pas très diversifiées ou contiennent des ordres dans leurs valeurs.

Troisième question : Sous-suites ordonnées et moyenne des déséquilibres

---

Question : Reprenez le même processus d'estimation du déséquilibre moyen d'arbres binaires de recherche, mais cette fois-ci avec des suites de nombres entiers qui contiennent des sous-suites ordonnées de longueurs variables. Les longueurs de ces sous-suites pourront être choisies aléatoirement ou bien de longueur fixe ou encore de longueurs croissantes ou décroissantes.

Réponse :
Nous définissons donc une fonction nous permettant de générer aléatoirement des arbres AVL avec des sous-suites ordonnées. Nous supposons que ces sous-suites de nombres ordonnés peuvent ne peuvent être contigües. En d'autres termes nous pouvons avoir, par exemple : [22 ; 45 ; 98 ; 1587]. Mais nous ne pouvons pas avoir [22 ; 11 ; 98 ; 1587]. De même nous supposons que nous devons toujours avoir le même ordre et que l'ordre choisi n'influencera pas notre étude. Aussi nous avons choisi un ordre croissant non contigüe. La fonction ci-dessous reflète ces choix :

```
1  let bst_rndSeries_create(treeSize, seriesLen : int * int) : int bst =
2
3    let randABR : int bst ref = ref (empty()) in
4    let fillerCount : int ref = ref treeSize in
5
6    (* Tant que l'arbre n'est pas rempli on génère des sous-suites d'entiers
       ordonnés
7       qu'on ajoute dans l'arbre: *)
8    while(!fillerCount > 0) do
9      let len : int = if(seriesLen<=0) then Random.int 101 else seriesLen in
10     let randLowerBound : int ref = ref (Random.int 101) in
11
12     (* On boucle jusqu'à la longueur de la sous-suite: *)
13     for i=1 to len do
14
15       (* Si l'arbre est rempli avant d'arriver au bout de la sous-suite, ce
         test
16           garantit qu'aucun ajout supplémentaire ne sera fait: *)
17       if(!fillerCount > 0)
18       then (
19         randABR := bst_linsert(!randABR, !randLowerBound);
20
21         (* On remplit l'arbre...    chaque ajout de noeud on décrémente: *)
22         fillerCount := !fillerCount - 1;
23
24         (* Nos entiers sont croissants mais choisis aléatoirement: *)
25         randLowerBound := Random.int(101) + !randLowerBound;
26       )
27     done;
28
29   done;
30
31   !randABR
32 ;;
```

Les fonctions ci-dessous sont alors des adaptations de l'exercice précédent à cette nouvelle fonction de génération :

```
let avgSeriesDesequilibre(sampleSize, treeSize, seriesLenMode : int * int * char) : float =

  let sum : float ref = ref 0. in

  for i=1 to sampleSize do
    match seriesLenMode with
    | 'r' -> sum := !sum +. float_of_int(desequilibre(bst_rndSeries_create(treeSize, -1)))
    | 'f' -> sum := !sum +. float_of_int(desequilibre(bst_rndSeries_create(treeSize, 10)))
    | 'a' -> sum := !sum +. float_of_int(desequilibre(bst_rndSeries_create(treeSize, i)))
    | 'd' -> sum := !sum +. float_of_int(desequilibre(bst_rndSeries_create(treeSize, sampleSize-i)))
    | _ -> failwith("Wrong mode for series length... 'r' for random lengths, 'f' for fixed lengths, 'a' for ascending lengths, and 'd' for descending lengths.");
  done;

  !sum /. float_of_int(sampleSize)
;;

let avgAvgSeriesDesequilibre(nbAvgs, sampleSize, treeSize, seriesLenMode : int * int * int * char) : float =

  let sum : float ref = ref 0. in

  for i=1 to nbAvgs do
    sum := !sum +. avgSeriesDesequilibre(sampleSize, treeSize, seriesLenMode)
  done;

  !sum /. float_of_int(nbAvgs)
;;
```

Nous obtenons alors, par exemple, avec des sous-suites de longueurs aléatoires, ces résultats :

```
# avgAvgSeriesDesequilibre(100, 100, 100, 'r');;
- : float = -63.9950000000000259
# avgAvgSeriesDesequilibre(100, 100, 100, 'r');;
- : float = -64.1277999999999651
# avgAvgSeriesDesequilibre(100, 100, 100, 'r');;
- : float = -64.092400000000012
```

Pour des sous-suites de longueurs croissantes :

```
# avgAvgSeriesDesequilibre(100, 100, 100, 'a');;
- : float = -49.5038
# avgAvgSeriesDesequilibre(100, 100, 100, 'a');;
- : float = -49.4555
# avgAvgSeriesDesequilibre(100, 100, 100, 'a');;
- : float = -49.4612000000000194
```

Pour des sous-suites de longueurs décroissantes :

```
# avgAvgSeriesDesequilibre(100, 100, 100, 'd');;
- : float = -49.1975999999999942
# avgAvgSeriesDesequilibre(100, 100, 100, 'd');;
- : float = -49.138600000000018
# avgAvgSeriesDesequilibre(100, 100, 100, 'd');;
- : float = -49.1900999999999868
```

Pour des sous-suites de longueurs fixes (la longueur choisie est 10 ici) :

```
# avgAvgSeriesDesequilibre(100, 100, 100, 'f');;
- : float = -11.0630999999999986
# avgAvgSeriesDesequilibre(100, 100, 100, 'f');;
- : float = -11.0670000000000019
# avgAvgSeriesDesequilibre(100, 100, 100, 'f');;
- : float = -11.0959999999999965
```

Conclusion : Ces données semblent indiquer que lorsque les données à insérer contiennent des sous-suites le déséquilibre est alors proportionnel à la moyenne de la longueur de ces sous-suites. Ce qui peut vite devenir un problème. En effet, avec seulement des longueurs de 10 éléments ordonnées on arrive à produire un déséquilibre dans nos ABR autour de 11 vers son sous-arbre droit. Ce qui peut potentiellement ralentir des opérations comme les opérations d'ajout, de suppression et de recherche dans l'ABR ainsi produit.

Ce-dernier point est d'ailleurs ce qui a motivé la conception des arbres AVL et que nous étudions dans la seconde partie de ce rapport.

Deuxième partie

Exercice 2 - Arbres AVL

Chapitre 1

Implantation d'un module Avl

Implantation des opérations de rotation

Question : Implantez les rotations à partir des axiomes et des exemples fournis.

Réponse :

Quelques utilitaires :

Pour répondre à ce problème et par anticipation de la question suivante nous allons définir quelques utilitaires :

```
1 type 'a avl = ('a * int) bst;;
```

Les arbres binaires AVL sont des types d'arbres binaires de recherches, ils utilisent donc la définition du type des arbres binaires de recherche (bst = binary search tree). Cependant, dans la question suivante il est demandé d'utiliser un stockage du déséquilibre, ceci afin d'éviter des appels à la fonction hauteur qui a une complexité en O(n), 'n' étant la taille de l'arbre [1]...

```
1  let getValue(tree : 'a avl) : 'a =
2    let (v, deseq) : ('a * int) = root(tree) in
3    v
4  ;;
5
6  let getHeight(tree : 'a avl) : int =
7    if(isEmpty(tree))
8    then 0
9
10   else let (v, h) : ('a * int) = root(tree) in h
11 ;;
```

1. En effet, nous cherchons à obtenir des complexités pour l'ajout et la suppression en $O(log_2(n))$. Un appel à une fonction de complexité linéaire ruinerait donc cet objectif.

Toutefois nous n'avons pas trouvé d'implantation satisfaisante des fonctions d'ajout et de suppression avec le stockage du déséquilibre dans les noeuds. En effet, les implantations que nous trouvions nous amenaient soit à nous éloigner des axiomes - ce qui ne nous permettait plus de garantir que nos fonctions étaient correctes - soit à réécrire des signatures de fonctions, ce qui ne répondait alors pas au cahier des charges.

Cependant nous avons trouvé une autre solution qui nous permet de rester au plus près des axiomes, de conserver les signatures et les complexités attendues. Cette solution consiste à stocker en chaque noeud la hauteur de son sous-arbre et non son déséquilibre.

Ainsi, les fonctions getValue et getHeight s'occupent de retourner respectivement la valeur contenue par le noeud de l'arbre (utile à un quelconque utilisateur de notre fonction) et la hauteur du sous-arbre en ce noeud (un détail d'implantation caché pour l'utilisateur).

```
1  let getNewHeight(tree : 'a avl) : 'a avl =
2    if(isEmpty(tree))
3    then tree
4
5    else
6        let ((v, h), g, d) : ('a * int) * 'a avl * 'a avl = (root(tree), lson
     (tree), rson(tree)) in
7        rooting( ( v, 1 + max(getHeight(lson(tree)), getHeight(rson(tree))) )
     , g, d)
8  ;;
9
10 let desequilibre(tree : 'a avl) : int =
11   if(isEmpty(tree))
12   then 0
13
14   else
15     getHeight(lson(tree)) - getHeight(rson(tree))
16 ;;
```

C'est notamment grâce à ces deux dernières fonctions ci-dessus que le stockage de la hauteur dans le noeud fonctionne. La première nous renvoie un arbre AVL dont la hauteur de l'arbre passé en paramètre a été réévalué. Cette fonction est appelée à chaque changement de la structure de l'arbre AVL (rotation, ajout, suppression...) c'est-à-dire à chaque rooting. Comme cette réévaluation s'appuie sur des stockages, sa complexité est celle de l'addition : O(1)

La seconde s'appuie sur la hauteur mise à jour à chaque modification de l'arbre pour calculer le déséquilibre en utilisant la même définition que dans les axiomes fournis.

Nos fonctions de rotations :

```
let rg(t : 'a avl) : 'a avl =
  if(isEmpty(t) || isEmpty(rson(t)))
  then failwith("Tree or right subtree is empty. Function rg can't continue
   ")

  else
    (
      let (p, u, (q, v, w)) : 'b * 'a avl * ('b * 'a avl * 'a avl) =
        (
          root(t),
          lson(t),
          (root(rson(t)), lson(rson(t)), rson(rson(t)))
        )
      in
      getNewHeight(rooting(q, getNewHeight(rooting(p, u, v)), w))
    )
;;



let rd(t : 'a avl) : 'a avl =
  if(isEmpty(t) || isEmpty(lson(t)))
  then failwith("Tree or right subtree is empty. Function rd can't continue
   ")

  else
    (
      let (q, (p, u, v), w) : 'b * ('b * 'a avl * 'a avl) * 'a avl =
        (
          root(t),
          (root(lson(t)), lson(lson(t)), rson(lson(t))),
          rson(t)
        )
      in
      getNewHeight(rooting(p, u, getNewHeight(rooting(q, v, w))))
    )
;;
```

Pour les fonctions de rotation gauche et de rotation droite nous suivons strictement les axiomes du cours. Le nommage des racines et des sous-arbres suivent ceux des schémas fournis dans les diapositives.

```
let rgd(t : 'a avl) : 'a avl =

  let (v, g, d) : 'b * 'a avl * 'a avl = (root(t), lson(t), rson(t)) in
  let t1 : 'a avl = rooting(v, rg(g), d) in
  let t_res : 'a avl = rd(t1) in

  t_res
;;


let rdg(t : 'a avl) : 'a avl =

  let (v, g, d) : 'b * 'a avl * 'a avl = (root(t), lson(t), rson(t)) in
  let t1 : 'a avl = rooting(v, g, rd(d)) in
  let t_res : 'a avl = rg(t1) in

  t_res
;;
```

Pour ces deux dernières fonctions nous nous appuyons sur une propriété du cours qui indique, par exemple, qu'une rotation gauche droite (fonction rgd() ici) est une rotation gauche du sous-arbre gauche suivie qu'une rotation droite de l'arbre passé en paramètre. Cela présente l'intérêt d'avoir une écriture réduite pour ces deux dernières fonctions.

Implantation de l'opération "rééquilibrer"

Question : Implantez l'opération reequilibrer à partir des axiomes et en supposant que le déséquilibre d'un arbre est stocké à la racine de cet arbre (proposez une manière de réaliser ce stockage à partir du type 'a t_btree)

Réponse :
En traduisant les axiomes du cours par le langage support Ocaml et en utilisant les principes et les fonctions outils définies précédemment nous obtenons alors la fonction de rééquilibrage suivante :

```
let reequilibrer(tree : 'a avl) : 'a avl =
  let deseq : int = desequilibre(tree) in

  if(deseq >= -1 && deseq <= 1)
  then tree

  else if(deseq = 2)
  then
    (
      let ldeseq : int = desequilibre(lson(tree)) in

      if(ldeseq = 1 || ldeseq = 0)
      then rd(tree)

      else rgd(tree)
    )

  else if(deseq = -2)
  then
    (
      let rdeseq : int = desequilibre(rson(tree)) in

      if(rdeseq = -1 || rdeseq = 0)
      then rg(tree)

      else rdg(tree)
    )

  else
    (
      failwith("reequilibrer: Something went really wrong,
                all nodes' unbalance values should be in {-2, -1, 0, 1, 2}"
    )
    )
;;
```

Implantation des opérations d'insertion et de suppression dans un arbre AVL

Question : Implantez les opérations d'insertion et de suppression dans un arbre AVL.

Réponse :

Notre fonction d'ajout :

```
let rec ajt_avl(e, tree : 'b * 'a avl) : 'a avl =
  if(isEmpty(tree))
  then rooting((e, 1), empty(), empty())

  else
    (
      let ((v, h), g, d) : (('a * int) * 'a avl * 'a avl) =
        ((getValue(tree), getHeight(tree)), lson(tree), rson(tree)) in

      if(e < v)
      then reequilibrer(getNewHeight(rooting((v, h), ajt_avl(e, g), d)))

      else if(e > v)
      then reequilibrer(getNewHeight(rooting((v, h), g, ajt_avl(e, d))))

      else tree
    )
;;
```

Notre fonction de suppression :

```
let rec avlDmax(tree : 'a avl) : 'a avl =
  if(isEmpty(tree))
  then failwith("Tree is empty, can't apply dmax function on an empty tree")

  else
    (
      let (v, g, d) : ('b * 'a avl * 'a avl) = (root(tree), lson(tree), rson(tree)) in

      if(isEmpty(d))
      then g

      else reequilibrer(getNewHeight(rooting(v, g, avlDmax(d))))
    )
;;


let rec suppr_avl(e, tree : 'b * 'a avl) : 'a avl =
  if(isEmpty(tree))
  then empty()

  else
    (
      let ((v, h), g, d) : (('a * int) * 'a avl * 'a avl) =
        ((getValue(tree), getHeight(tree)), lson(tree), rson(tree)) in

      if(e < v)
      then reequilibrer(getNewHeight(rooting((v, h), suppr_avl(e, g), d)))

      else if(e > v)
      then reequilibrer(getNewHeight(rooting((v, h), g, suppr_avl(e, d))))

      else
        (
          if(isEmpty(d))
          then g

          else if(isEmpty(g))
          then d

          else reequilibrer(getNewHeight(rooting(bstMax(g), avlDmax(g), d)))
        )
    )
;;
```

L'utilitaire avlDmax était aussi indiqué dans les axiomes. L'implantation choisie est là encore une traduction quasi immédiate des axiomes du cours.

Opération de recherche dans un AVL

Question : Pouvez-vous réutiliser l'opération de recherche du module Bst ? Si ce n'est pas le cas modifiez votre code afin de pouvoir le faire.

Réponse :

```
let rec seek_avl(v,b : 'a * 'a avl) : 'a avl =

  if(isEmpty(b))
  then b

  else
    (
      let (fg,fd) : ('a avl * 'a avl) = (lson(b),rson(b))
      in

      if(v = getValue(b))
      then b

      else if(v < getValue(b))
      then seek_avl(v, fg)

      else seek_avl(v, fd)
    )
;;
```

La fonction bst_seek du module des arbres binaires de recherche ne fonctionnant plus à cause du stockage à la racine de chaque noeud nous avons dû redéfinir une fonction de recherche très inspiré de celle du module des arbres binaires de recherche mais qui tient compte de ce stockage pour l'ignorer.

Chapitre 2

Expérimentations avec les arbres AVL

Génération aléatoire d'AVL et complexités des opérations implantées

---

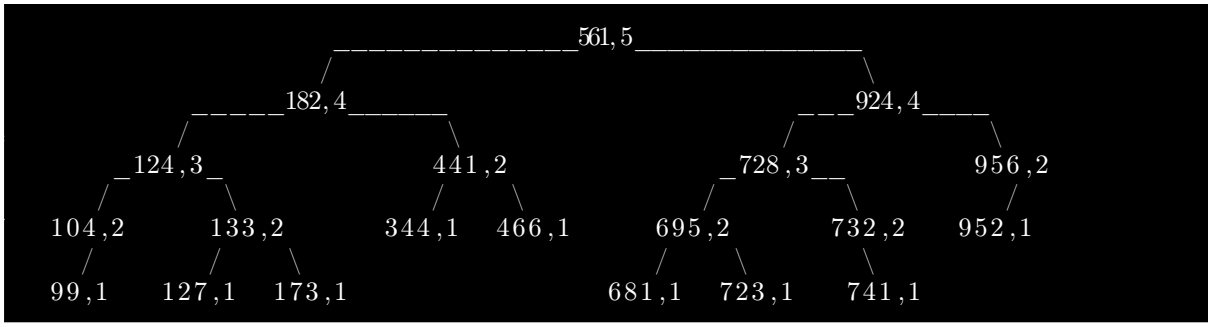
Question : Définissez une fonction avl_rnd_create qui crée des arbres AVL à partir de suites d'entiers aléatoires et vérifiez expérimentalement que les opérations de recherche, d'insertion et de suppression ont bien une complexité en $\theta(logn)$ où n est la taille de l'arbre.

Réponse :

avl_rnd_create, un autre utilitaire :

```
1  let avl_rnd_create(bound, treeSize : int * int) : 'a avl =
2
3    let randABR : 'a avl ref = ref (empty()) in
4
5    for i=1 to treeSize do
6      let randInt : int = Random.int bound in
7      randABR := ajt_avl(randInt, !randABR);
8    done;
9
10   !randABR
11 ;;
```

Cette fonction initialise une suite d'entiers pseudo-aléatoires, crée un arbre avl mutable vide et s'occupe de remplir cet arbre mutable vide en fonction d'une taille d'arbre passée en paramètre. Le paramètre bound permet simplement de définir une valeur que les valeurs aux noeuds ne peuvent pas dépasser.



Listing 2.1 – Exemple d'arbres AVL de taille 20 généré par la fonction avl_rnd_create

Maintenant que nous avons une fonction qui nous permet de générer aléatoirement des AVL nous pouvons évaluer la complexité de nos opérations.

Pour cela nous utilisons deux familles de fonctions :

```
let chrono(func, args : ('a -> 'b) * ('c * 'c avl)) : float =
  let start_chrono : float = Sys.time() in
  ignore(func(args));
  Sys.time() -. start_chrono
;;

let avgAvlOp(avlOp, sampleSize, treeSize : (('a * 'a avl) -> 'c) * int *
    int) : float =

  let dummyAVL : 'a avl = avl_rnd_create(10000000, treeSize) in
  let sum : float ref = ref 0. in

  for i=1 to sampleSize do
    let randInt : int = Random.int 10000000 in
    sum := !sum +. chrono(avlOp, (randInt, dummyAVL));
  done;

  !sum /. float_of_int(sampleSize)
;;
```

La famille 'chrono', définie dans le module avlExperimentsUtils.ml du dossier modules_etu de notre projet, nous permet de mesurer le temps d'exécution d'une fonction sur nos avl quelconque avec son tuple de paramètres souhaité. Au préalable nous avons généré un arbre. Un exemple d'utilisation est le suivant :

```
let otherdummy : 'a avl = avl_rnd_create(1000, 30);;
avgAvlOp(ajt_avl, 100000, 256);;
```

L'autre fonction est une famille de fonction : ajtGraph, supprGraph et seekGraph. Elles génèrent des graphiques de complexités pour nos trois fonctions sur les AVL. Elles s'appuient sur des modules fournis en L2 par l'université de Poitiers. Leurs définitions se trouvent dans le module avlGraphicsUtils.ml dans le dossier modules_etu\avl...

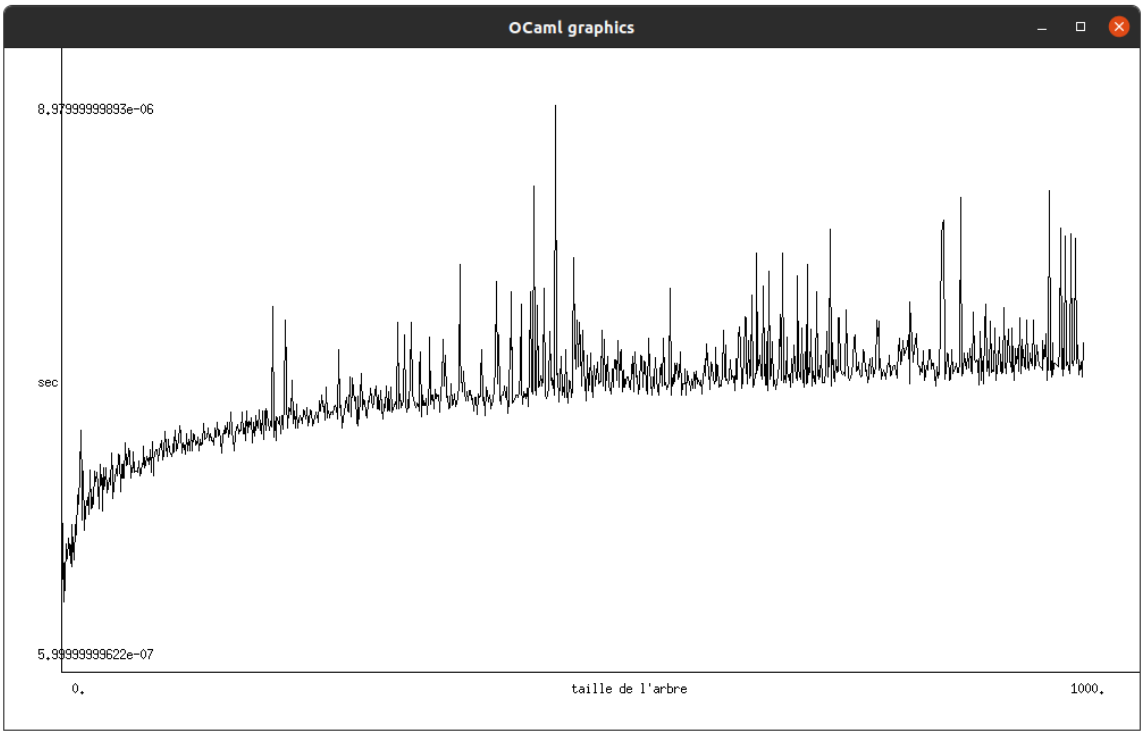
Résultats de nos expérimentations :

Pour l'ajout :

1) Mesures avec la fonction chrono :

Taille de l'arbre	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Temps d'exécution	2.00e-06	2.21e-06	2.43e-06	2.87e-06	3.58e-06	3.85e-05	4.31e-06	4.69e-06

2) Mesure avec ajtGraph :

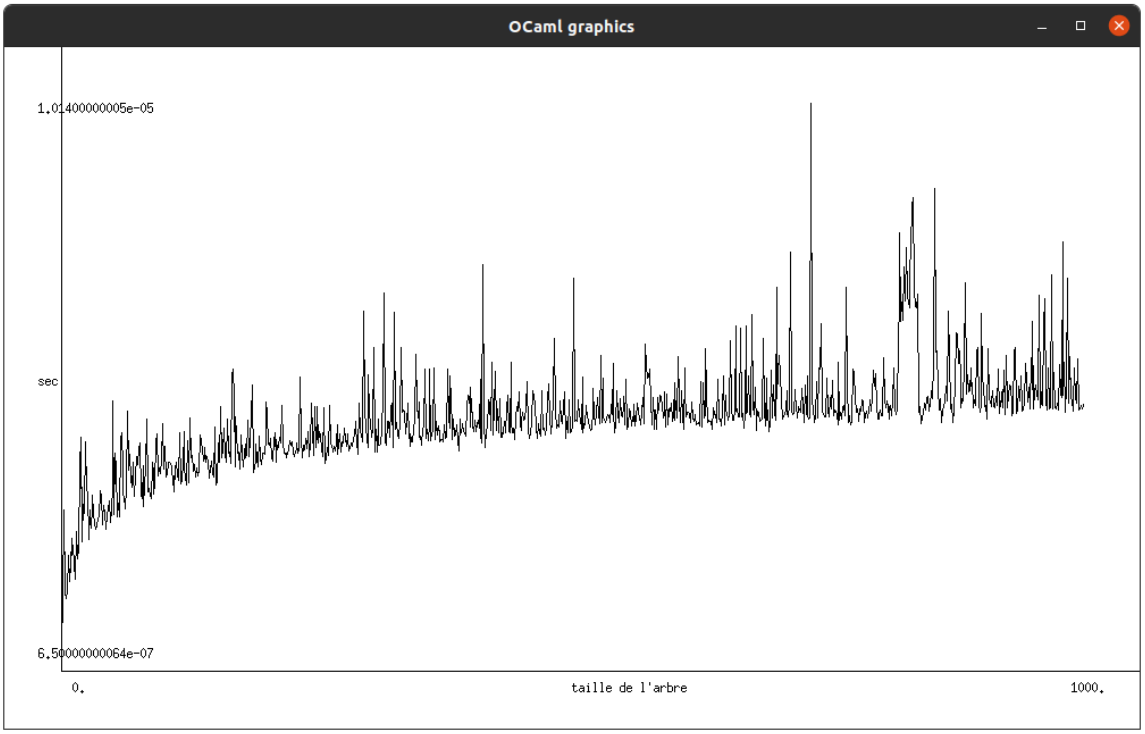


Pour la suppression :

1) Mesures avec la fonction chrono :

Taille de l'arbre	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Temps d'exécution	1.36e-06	1.60e-06	2.01e-06	2.47e-06	2.99e-06	3.35e-06	3.78e-06	4.22e-06

2) Mesure avec supprGraph :

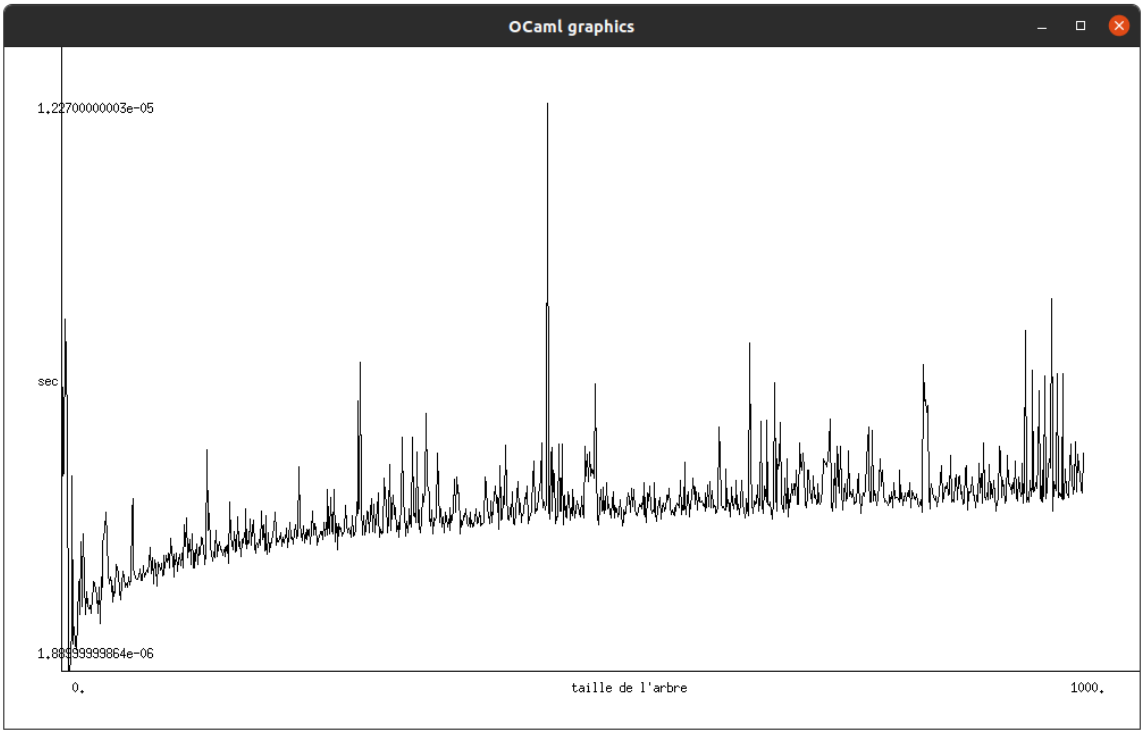


Pour la recherche :

1) Mesures avec la fonction chrono :

Taille de l'arbre	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8
Temps d'exécution	7.47e-07	8.56e-07	9.52e-07	1.08e-06	1.13e-06	1.23e-06	1.32e-06	1.42e-06

2) Mesure avec seekGraph :



Sous-suites ordonnées et nombre de rotations dans un AVL

Question : En créant des arbres AVL avec des suites de nombres entiers qui contiennent des sous-suites ordonnées de longueur variable, estimez le nombre moyen de rotations qui sont effectuées pour garder l'arbre équilibré. Comment ce nombre évolue-t-il en fonction de la taille de l'arbre ?

Réponse :
Pour réaliser cette estimation nous nous sommes appuyés sur quelques utilitaires :

```
let nbRots : int ref = ref 0;;
```

Tout d'abord une variable globale mutable. Nous avons cherché à faire au plus simple. Ce n'est certes pas le plus élégant mais c'est ce qui était le plus facile pour nous pour confirmer la validité des résultats que nous obtiendrions...

```
let reequilibrer_stats(tree : 'a avl) : 'a avl =
  let deseq : int = desequilibre(tree) in

  if(deseq >= -1 && deseq <= 1)
  then tree

  else if(deseq = 2)
  then
    (
      let ldeseq : int = desequilibre(lson(tree)) in

      if(ldeseq = 1 || ldeseq = 0)
      then (
        nbRots := !nbRots + 1;
        rd(tree)
      )

      else
        (
          nbRots := !nbRots + 1;
          rgd(tree)
        )
    )

  else if(deseq = -2)
  then
    (
      let rdeseq : int = desequilibre(rson(tree)) in

      if(rdeseq = -1 || rdeseq = 0)
      then (
        nbRots := !nbRots + 1;
        rg(tree)
      )

      else (
        nbRots := !nbRots + 1;
```

```
38          rdg(tree)
39        )
40      )
41
42    else
43      (
44        failwith("reequilibrer: Something went really wrong,
45                  all nodes' unbalance values should be in {-2, -1, 0, 1, 2}"
      )
46      )
47  ;;
```

À chaque rotation nous incrémentons cette variable globale d'un. L'inconvénient de cette démarche, nous le voyons ici, est que nous avons dû redéfinir les fonctions rééquilibrer et ajout pour tenir compte de l'incrémentation de la variable globale.

De manière analogue à l'exercice 1 nous avons alors défini une fonction de génération d'arbres AVL avec des sous-suites ordonnées comme suit :

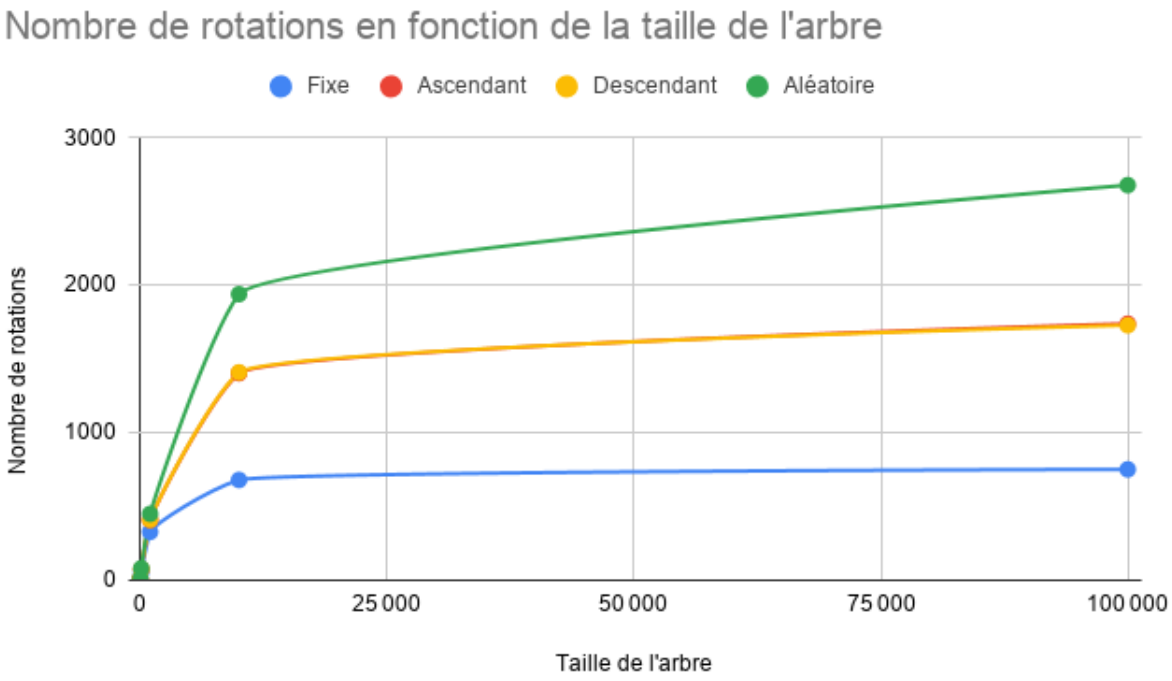
```
1  let avl_rndSeries_create(treeSize, seriesLen : int * int) : int =
2
3    let randAVL : int avl ref = ref (empty()) in
4    let fillerCount : int ref = ref treeSize in
5
6    (* Tant que l'arbre n'est pas rempli on génère des sous-suites d'entiers
       ordonnés
7       qu'on ajoute dans l'arbre: *)
8    while(!fillerCount > 0) do
9      let len : int = if(seriesLen<=0) then Random.int 101 else seriesLen in
10     let randLowerBound : int ref = ref(Random.int 1001) in
11
12     (* On boucle jusqu'à la longueur de la sous-suite: *)
13     for i=1 to len do
14
15       (* Si l'arbre est rempli avant d'arriver au bout de la sous-suite, ce
         test
16          garantit qu'aucun ajout supplémentaire ne sera fait: *)
17       if(!fillerCount > 0)
18       then (
19
20         (* ajt_avl_stats compte le nombre de rotations qui a eu lieu lors
       de l'ajout:  *)
21         randAVL := ajt_avl_stats(!randLowerBound, !randAVL);
22
23         (* On remplit l'arbre...    chaque ajout de noeud on décrémente: *)
24         fillerCount := !fillerCount - 1;
25
26         (* Nos entiers sont croissants mais choisis aléatoirement: *)
27         randLowerBound := Random.int(101) + !randLowerBound;
28       )
29     done;
30
31   done;
32
33   !nbRots
34 ;;
```

En nous appuyant sur cette dernière fonction nous avons alors créé une fonction qui évalue la moyenne du nombre de rotations effectuées pour générer des arbres de tailles données. Cette fonction admet 4 modes : un mode où la longueur des sous-suites est fixe, un mode où cette longueur est croissante, décroissante et enfin aléatoire. Cette fonction est la suivante :

```
let ajtNbRotsAvg(sampleSize, treeSize, seriesLenMode : int * int * char) :
    float =

 let sum : float ref = ref 0. in

 for i=1 to sampleSize do
  nbRots := 0;
  match seriesLenMode with
  | 'r' -> sum := !sum +. float_of_int(avl_rndSeries_create(treeSize, -1))
  | 'f' -> sum := !sum +. float_of_int(avl_rndSeries_create(treeSize, 10))
  | 'a' -> sum := !sum +. float_of_int(avl_rndSeries_create(treeSize, i))
  | 'd' -> sum := !sum +. float_of_int(avl_rndSeries_create(treeSize,
    sampleSize-i))
  | _ -> failwith("Wrong mode for series length... 'r' for random lengths,
    'f' for fixed lengths, 'a' for ascending lengths, and 'd' for descending
     lengths.");
 done;

 !sum /. float_of_int(sampleSize)
;;
```

Nous avons maintenant tous les outils pour réaliser nos estimations... Nous les avons compilé dans le tableau et graphe associé suivants :

Taille de l'arbre	10^1	10^2	10^3	10^4	10^5
Longueurs fixes	5.91	49.34	322.88	675.37	747.34
Longueurs croissantes	5.81	67.82	403.92	1400.04	1736.08
Longueurs décroissantes	5.77	68.82	405.02	1405.95	1725.62
Longueurs aléatoires	5.87	75.6	445.25	1936.7	2676.71



Conclusion : Ce tableau et ce graphe nous permettent ainsi de conclure que le nombre de rotations requis pour générer des arbres AVL de taille 'n' croît logarithmiquement avec la taille de l'arbre.
