Rapport d’expériences

05 février 2021

Auteurs : Fréjoux Gaëtan, Niord Mathieu

Sommaire

Partie A - Expérimentations sur les ABR

1. Déséquilibre moyen

Pour calculer le déséquilibre moyen d'un arbre binaire de recherche à partir de valeurs générées aléatoirement nous avons utilisé les fonctions suivantes :

Fonctions :

```
(*Fonction qui crée un arbre binaire de recherche avec des valeurs
aléatoires*)
let rec bst_rnd_create(sizeTree, nbMax : int * int) : int bs_tree =
  if(sizeTree = 0)
  then empty()
  else let t = bst_rnd_create((sizeTree-1),nbMax) in
       bst_linsert(t,Random.int nbMax)
;;
```

```
(*Fonction qui renvoie la hauteur d'un arbre binaire*)
let rec height(tree : 'a t_btree) : int =
  if(isEmpty(tree) || (isEmpty(lson(tree)) && isEmpty(rson(tree))))
  then 0
  else 1 + max (height(lson(tree))) (height(rson(tree)))
;;
```

```
(*Fonction qui calcule le déséquilibre d'un arbre binaire de recherche*)
let desequilibre(nbTree, sizeTree, nbMax : int * int * int) : float =
  let somme : float ref = ref 0. and
      t : int bs_tree ref = ref (empty()) in
  for i = 0 to nbTree do
    t := bst_rnd_create(sizeTree, nbMax);
    somme := !somme +. float_of_int(height(lson(!t)) -
height(rson(!t)));
  done;
  (!somme)/.float_of_int(nbTree)
;;
```

Tests :

```
# desequilibre(1, 10, 100000);;
- : float = -5.
# desequilibre(10, 10, 100000);;
- : float = -1.4
# desequilibre(100, 100, 100000);;
- : float = -0.52
# desequilibre(1000, 1000, 100000);;
- : float = -0.253
# desequilibre(10000, 1000, 100000);;
- : float = 0.0018
```

Nous avons observé que plus l'arbre est grand, plus le déséquilibre tend vers 0. On peut en déduire que pour un arbre infiniment grand, il n'y aura pas de déséquilibre (ou peu).

2. Déséquilibre avec des sous-suites :

Fonctions :

```
(*Fonction qui construit un abr selon un mode choisi
  mode -1 : avec liste décroissante
  mode 0  : avec liste de même longueur
  mode 1  : avec liste croissante
  mode 2  : avec liste de taille aléatoire
 *)
let bst_list_create(mode, nbList, nbElement, nbMax) : int bs_tree =
  Random.self_init ();
  let answer : int bs_tree ref = ref (empty()) and
      listLen : int ref = ref nbList and
      random : int ref  = ref (Random.int nbMax) and
      elemLen : int ref = ref nbElement in
  if(mode < 3 && mode > -2)
  then (
    while(!listLen > 0) do
      (
        (*Dans le cas où le mode choisi est celui de taille aléatoire,
            la taille de la liste est défini juste ici :*)
        if(mode = 2)
        then elemLen := Random.int nbElement;
        for i = 0 to !elemLen do
```

```
          answer :=  bst_linsert(!answer, !random);
          random := !random + Random.int 100;
        done;
        random :=  Random.int nbMax;
        elemLen := !elemLen + mode;
        listLen:= !listLen - 1;
      )
    done;
    !answer
  )
  else rooting(0,empty(),empty())
;;
```

```
let list_desequilibre(mode, nbList, nbElement,nbMax ,nbTree) : int =
  let somme : int ref = ref 0 and
      t : int bs_tree ref = ref (empty()) in
  for i = 0 to nbTree do
    t := bst_list_create(mode, nbList, nbElement,nbMax);
    somme := !somme + (height(lson(!t)) - height(rson(!t)));
  done;
  (!somme)/nbTree
;;
```

Tests :

```
# list_desequilibre(-1, 100, 100, 1000000, 100);;
- : int = -130
# list_desequilibre(0, 100, 100, 1000000, 100);;
- : int = -99
# list_desequilibre(1, 100, 100, 1000000, 100);;
- : int = -12
# list_desequilibre(2, 100, 100, 1000000, 100);;
- : int = -12
# list_desequilibre(-1, 100, 100, 1000000, 100);;
- : int = -146
# list_desequilibre(0, 100, 100, 1000000, 100);;
- : int = -50
# list_desequilibre(1, 100, 100, 1000000, 100);;
- : int = -164
# list_desequilibre(2, 100, 100, 1000000, 100);;
- : int = -17
```

Nous remarquons, lors des divers cas d'utilisation de sous-suites pour remplir un arbre, qu'il n'est pas possible de générer un arbre équilibré. Les résultats obtenus sont aléatoires, cependant le mode aléatoire (ici 2) est celui avec lequel nous nous rapprochons le plus d'un résultat (arbre) équilibré.

Partie B - Expérimentations sur les AVL

1. Fonction d'ajout

De manière à expérimenter notre méthode d'ajout, nous avons décidé de l'appliquer sur un nombre donné d'arbres de 2^(n) valeurs, toutes bornées par une valeur maximum, générés aléatoirement. Enfin, nous mesurons le temps d'exécution lors de l'ajout d'une valeur supérieure à la borne (qui sera par conséquent supérieure à l'ensemble des valeurs de l'AVL généré).

Fonction :

```
  let  averageTimePerPowerAdd  (nbMax,power,nbTime  :  int  *float*int)  :
float  =
  let tree : int avl ref = ref (empty()) in
  let time : float ref = ref 0. in
  let somme : float ref = ref 0. in
  for i = 1 to nbTime do
    tree := avl_rnd_create(nbMax, int_of_float((2. ** power)));
    time := Sys.time();
    tree := ajt_avl((nbMax+200),!tree);
    time := Sys.time() -. !time;
    somme := !somme +. !time;
  done;
  ((!somme)/.float_of_int(nbTime))
;;
```

Comme on le voit ci-dessus, notre fonction de test prend trois attributs qui correspondent (respectivement) à : la borne, la puissance et le nombre de fois que l'on souhaite réaliser ce

test. Cette fonction nous permet d'observer la moyenne de temps d'ajout pour une puissance donnée.

Tests :

```
# averageTimePerPowerAdd(1000,1.,1000);; ← 2 valeurs
- : float = 2.69200000005831421e-06
# averageTimePerPowerAdd(1000,2.,1000);; ← 4 valeurs
- : float = 2.99099999999263826e-06
# averageTimePerPowerAdd(1000,3.,1000);; ← 8 valeurs
- : float = 3.16799999997385844e-06
# averageTimePerPowerAdd(1000,4.,1000);; ← 16 valeurs
- : float = 3.95300000000276e-06
# averageTimePerPowerAdd(1000,5.,1000);; ← 32 valeurs
- : float = 4.60599999999367058e-06
# averageTimePerPowerAdd(1000,6.,1000);; ← 64 valeurs
- : float = 4.43799999997374082e-06
# averageTimePerPowerAdd(1000,7.,1000);; ← 128 valeurs
- : float = 5.02399999997038571e-06
# averageTimePerPowerAdd(1000,8.,1000);; ← 256 valeurs
- : float = 5.31799999999371645e-06
# averageTimePerPowerAdd(1000,9.,1000);; ← 512 valeurs
- : float = 5.78399999997891737e-06
# averageTimePerPowerAdd(1000,10.,1000);; ← 1024 valeurs
- : float = 6.42900000003265882e-06
```

Comme on peut le voir ci-dessus, lors de chaque test, le nombre de valeurs stockées dans l'arbre double, et que contrairement aux valeurs, les résultats augmentent légèrement par rapport aux précédents. On observe une complexité logarithmique pour la fonction d'ajout.

2. Fonction de suppression

Comme nous l'avons fait avec la méthode d'ajout, pour tester notre fonction de suppression, nous avons décidé de l'appliquer sur un nombre donné d'arbres de 2^(n) - 1 valeurs, toutes bornées par une valeur maximum, générés aléatoirement, puis d'ajouter une dernière valeur supérieure au maximum pour mesurer le temps d'exécution de sa suppression.

Fonction :

```
let averageTimePerPowerSup (nbMax,power,nbTime : int *float*int) : float
=
  let tree : int avl ref = ref (empty()) in
  let time : float ref = ref 0. in
  let somme : float ref = ref 0. in
  for i = 1 to nbTime do
    tree := avl_rnd_create(nbMax, (int_of_float((2. ** power)))-1);
    tree := ajt_avl((nbMax+200),!tree);
    time := Sys.time();
    tree := sup_avl((nbMax+200),!tree);
    time := Sys.time() -. !time;
    somme := !somme +. !time;
  done;
  ((!somme)/.float_of_int(nbTime))
;;
```

Notre fonction de test prend trois attributs qui correspondent (respectivement) à : la borne, la puissance et le nombre de fois que l'on souhaite réaliser ce test. Cette fonction nous permet d'observer la moyenne de temps de la suppression pour une puissance donnée.

Tests :

```
# averageTimePerPowerSup(1000,0.,1000);;
- : float = 1.04399999993987091e-06
# averageTimePerPowerSup(1000,1.,1000);;
- : float = 1.7590000000176075e-06
# averageTimePerPowerSup(1000,2.,1000);;
- : float = 2.41700000002964563e-06
# averageTimePerPowerSup(1000,3.,1000);;
- : float = 2.97199999996777811e-06
# averageTimePerPowerSup(1000,4.,1000);;
- : float = 3.13000000005203545e-06
# averageTimePerPowerSup(1000,5.,1000);;
- : float = 3.47500000001232917e-06
# averageTimePerPowerSup(1000,6.,1000);;
- : float = 3.71900000000735552e-06
# averageTimePerPowerSup(1000,7.,1000);;
- : float = 4.23499999997645175e-06
# averageTimePerPowerSup(1000,8.,1000);;
- : float = 4.77900000000275318e-06
```

```
# averageTimePerPowerSup(1000,9.,1000);;
- : float = 5.53599999998155588e-06
# averageTimePerPowerSup(1000,10.,1000);;
- : float = 6.28400000000439726e-06
```

Comme on a pu le voir avec notre fonction d'ajout, ci-dessus nous remarquons que lors de chaque test, le nombre de valeurs stockées dans l'arbre double, et que contrairement aux valeurs, les résultats augmentent légèrement par rapport aux précédents. On observe une complexité logarithmique pour la fonction d'ajout.

3. Fonction de recherche

De la même manière qu'avec les fonctions de test précédentes, nous avons choisi d'appliquer notre fonction de recherche sur une valeur ajoutée en dernier dans un arbre (donc une valeur supérieure à toutes les autres valeurs de l'arbre). Le test s'effectue sur un nombre donné d'arbres générés aléatoirement à partir de 2^(n) - 1 valeurs aléatoires bornées.

Fonction :

```
let averageTimePerPowerSeek (nbMax,power,nbTime : int *float*int) :
float  =
  let tree : int avl ref = ref (empty()) in
  let time : float ref = ref 0. in
  let somme : float ref = ref 0. in
  for i = 1 to nbTime do
    tree := avl_rnd_create(nbMax, (int_of_float((2. ** power)))-1);
    tree := ajt_avl((nbMax+200),!tree);
    time := Sys.time();
    tree := seek_avl((nbMax+200),!tree);
    time := Sys.time() -. !time;
    somme := !somme +. !time;
  done;
  ((!somme)/.float_of_int(nbTime))
;;
```

Cette fonction prend trois attributs : une borne, une puissance et le nombre de fois que l'on souhaite réaliser ce test. Cela nous permet d'établir une moyenne de temps d'exécution d'une recherche pour une puissance donnée.

Tests :

```
# averageTimePerPowerSeek(1000,0.,1000);;
- : float = 8.36999999997090115e-07
# averageTimePerPowerSeek(1000,1.,1000);;
- : float = 1.14499999999306814e-06
# averageTimePerPowerSeek(1000,2.,1000);;
- : float = 1.34500000000059572e-06
# averageTimePerPowerSeek(1000,3.,1000);;
- : float = 1.57799999998697874e-06
# averageTimePerPowerSeek(1000,4.,1000);;
- : float = 1.80500000001515562e-06
# averageTimePerPowerSeek(1000,5.,1000);;
- : float = 1.84499999999143683e-06
# averageTimePerPowerSeek(1000,6.,1000);;
- : float = 2.00099999998482031e-06
# averageTimePerPowerSeek(1000,7.,1000);;
- : float = 2.18000000001961081e-06
# averageTimePerPowerSeek(1000,8.,1000);;
- : float = 2.44800000000644518e-06
# averageTimePerPowerSeek(1000,9.,1000);;
- : float = 2.87599999999876849e-06
# averageTimePerPowerSeek(1000,10.,1000);;
- : float = 3.39900000003545678e-06
```

Comme précédemment, le nombre de valeurs stockées double quand les résultats en temps augmente légèrement en comparaison avec les résultats précédents. On remarque ici aussi une complexité logarithmique de la fonction de recherche.

4. Nombre de rotations avec des sous-suites

Dans le cadre de notre expérience, nous avons dû définir les fonctions de rééquilibrage et d'ajout pour qu'elles renvoient en plus d'un arbre, un entier qui correspond à une rotation.

Fonctions :

```
let averageRotation(mode,nbList,nbElement,nbMax,nbTime : int * int * int
* int * int) : int * int=
  let rotation : int ref = ref 0 in
  let nbE = ref nbElement in
  let tree = ref (empty()) in
  let nb_valeur = ref 0 in
  if mode> -2 && mode <3
  then
    (
      for i = 1 to nbTime do
        for j = 1 to nbList do
          let randomV = ref (Random.int nbMax) in
          if(mode = 2)
          then nbE := Random.int (nbElement+1);
          for k = 1 to !nbE do
            let newT,r = ajt_avl(!randomV,!tree) in
            tree := newT;
            randomV := !randomV + Random.int 100;
            rotation := !rotation + r;
            nb_valeur := !nb_valeur +1;
          done;
          nbE := !nbE + mode;
        done;
        nbE := nbElement;
        tree := empty();
      done;

      (!rotation/nbTime),(!nb_valeur/nbTime)
    )
  else failwith "Error, wrong mode (averageRotationPerPower)"
;;
```

Cette fonction prend 5 paramètres, il y a le mode, le nombre de listes, le nombre d'éléments dans la première liste, la valeur maximale possible pour un nœud et le nombre de fois que l'on souhaite tester sur le même mode.

La fonction renvoie 2 entiers : le premier correspond au nombre moyen de rotations et le deuxième au nombre moyen de valeurs dans un arbre.

Tests :

```
# averageRotation(-1,100,100,10000000,50);;
- : int * int = (4898, 5050)
# averageRotation(0,100,100,10000000,50);;
- : int * int = (9703, 10000)
# averageRotation(1,100,100,10000000,50);;
- : int * int = (14428, 14950)
# averageRotation(2,100,100,10000000,50);;
- : int * int = (4814, 4975)
# averageRotation(-1,10,1000,10000000,50);;
- : int * int = (9737, 9955)
# averageRotation(0,10,1000,10000000,50);;
- : int * int = (9799, 10000)
# averageRotation(1,10,1000,10000000,50);;
- : int * int = (9825, 10045)
# averageRotation(2,10,1000,10000000,50);;
- : int * int = (4825, 4904)
```

De manière générale, on observe qu'il y a en moyenne une rotation par nœud.