Chapitre 5 - Arbres

Georges-Pierre BONNEAU (cours) - Mica MURPHY (note) - Antoine SAGET (note)

Lundi 3 Décembre 2018

Pourquoi faire des arbres (triés) ?

Coûts avantageux :

- Insertion : $O(1)$
- Suppresion : $O(1)$
- Recherche : $O(log(N))$

Structure de données hiérarchique :

- les objets sont stockés dans des nœuds
- nœud spécial : le nœud racine, tout en haut de la hiérarchie
- tous les autres noœuds n'ont qu'un seul nœud parent
- tous les nœuds (y compris la racine) peuvent avoir 0, 1 ou plusieurs nœuds enfants
- les nœuds sans enfants sont appellés des feuilles de l'arbre
- chaque nœud peut être associé à une profondeur

Un arbre binaire

C'est un arbre dans lequel chaque nœud a au plus 2 nœuds enfants.

Représentation chaînée des arbres binaires à l'aide de pointeurs

À chaque objet on associe 2 pointeurs, un vers l'enfant "à gauche", un vers l'enfant "à droite".

- Type Objet : quelconque
- Triplet : le type <O : un objet ; gauche, droite : un AdTriplet>
- AdTriplet : le type pointeur vers un Triplet

Représentation chaînée des arbres n-aires

Triplet : le type <0 : un objet ; Enfant1, Soeur : un adTriplet >

Parcours des abres binaires

3 parcours récursifs : préfixé, post fixé, infixé (= symétrique)

Parcours préfixé

```
ParcoursPréFixé : action (donnée A : un AdTriplet)

Algo :
  Si A != Nil alors
    Afficher(Objet de A↑)
    ParcoursPréFixé(gauche de A↑)
    ParcoursPréFixé(droite de A↑)
```

Parcours postfixé

```
ParcoursPostFixé : action (donnée A : un AdTriplet)

Algo :
  Si A != Nil alors
    ParcoursPostFixé(gauche de A↑)
    ParcoursPostFixé(droite de A↑)
    Afficher(Objet de A↑)
```

Parcours infixé

```
ParcoursInFixé : action (donnée A : un AdTriplet)

Algo :
  Si A != Nil alors
    ParcoursInFixé(gauche de A↑)
    Afficher(Objet de A↑)
    ParcoursInFixé(droite de A↑)
```

Exemples

```
   3
 6   1
2 4 . 5
```

- PréFixé : **3** 6 2 4 | 1 5 (on commence par le haut)
- PostFixé : 2 4 6 | 5 1 **3** (on finit par le haut)
- InFixé : 2 6 4 **3** 1 5 (on passe par le haut à la moitié)

PostFixé :

```
(i,j) PoF(g, n, d) = (profondeur de récursion, i-ème appel à PoF) PoF(fils gauche, noœud, fils droit)

(1,1) PoF(6, 3, 1)
  (2,2) PoF(2, 6, 4)
    (3,3) PoF(., 2, .)
      (4,4) PoF( . )
      (5,4) PoF( . )
    Affiche(2)
    (6,3) PoF(., 4, .)
      (7,4) PoF( . )
      (8,4) PoF( . )
    Affiche(4)
  Affiche(6)
  (9,2) PoF(., 1, 5)
    (10,3) PoF( . )
    (11,3) PoF(., 5, .)
      (12,4) PoF( . )
      (13,4) PoF( . )
    Affiche(5)
  Affiche(1)
Affiche(3)
```

Algorithmes

Profondeur d'un nœud

```
ParcoursPréFixé : action(donnée 1 : un AdTriplet, donnée P : un entier)
  Affiche(Objet de A↑) // P la profondeur du nœud A
  ParcoursPréFixé(gauche de A↑, P+1)
  ParcoursPréFixé(droit de A↑, P+1)
```

Nombre de nœuds d'un arbre binaire

```
NbNœud : fonction(A : un AdTriplet) -> un entier positif

Algo :
  Si A = Nil retourne 0
  Sinon retourne 1 + NbNœud(gauche de A↑) + NbNœud(doite de A↑)
```

Nombre de feuilles d'un arbre

```
NbFeuille : fonction(donnée A : un AdTriplet) -> un entier

Algo :
  Si A = nil retourne 0
  Sinon
    Ng <- NbFeuille(gauche de A↑)
    Nd <- NbFeuille(droit de A↑)
    Si Ng = 0 et Nd = 0 alors retourner 1
    Sinon retourne Ng + Nd
```

Calcul de la profondeur maximum d'un nœud

```
ProfondeurMax : fonction(donnée A : un AdTriplet) -> un entier

Algo :
  Si A = Nil retourne 0
  Si A != Nil retourne 1 + Max(ProfondeurMax(gauche de A↑), ProfondeurMax(droite de A↑))
```

Parcours par niveaux

On s'aide d'une liste de noœuds de l'arbre : on va vouloir rajouter des nœuds en **début** et en **fin** de liste.

- T : pointeur vers le 1er élément de la liste
- Q : pointeur vers le dernier élément de la liste

1. On affiche la tête de liste
2. On rajoute ne fin de liste les nœuds enfants de la tête de liste
3. On supprime la tête de liste
4. On recommence jusqu'à ce que la liste soit vide

```
Objet : type quelconque (les objets dans les nœuds)

Triplet : <O : un objet ; gauche, droit : AdTriplet>

AdTriplet : type pointeur vers un Triplet

Doublet : <Info : un AdTriplet; suc : un AdDoublet>

Ad Doublet : type pointeur vers un Doublet
```

ParcoursParNiveau : action(donnée A : un AdTriplet)

```
Lexique :
  X, T, Q : un AdDoublet
  Courant : un AdTriplet

Algo :
  Si A != Nil
    allouer(T)
    Info de T↑ <- A
    Q <- T

    Itérer...
      // 1
      Courant <- Info de T↑
      Affiche(Objet de Courant↑)

      // 2
      Si gauche de Courant↑ != Nil :
        allouer(succ de Q↑)
        Q <- succ de Q↑
        Info de Q↑ <- gauche de Courant↑
      Si droite de Courant↑ != Nil :
        allouer(succ de Q↑)
        Q <- succ de Q↑
        Info de Q↑ <- droit de Courant↑
    ...Tant que T != Q

    // 3
    X <- T
    T <- succ de T↑
    Libérer(X)
```

Arbre binaire de recherche (ABR)

Définition

Objectif : accélérer la recherche (si il existe une **relation d'ordre** entre les éléments)

- C'est un **arbre binaire**
- L'arbre vide est un ABR
- **Clé** d'un objet : un attribut entier de chaque objet
- Un arbre non vide A est un ABR ssi
  - gauche de A↑ et droit de $A\hat{}$ sont des ABR
  - $\forall g$ nœud dans le sous-arbre gauche, clé de $g\hat{} \leq$ clé de $A\hat{}$
  - $\forall d$ nœud dans le sous-arbre droit, clé de $d\hat{} >$ clé de $A\hat{}$

Recherche d'une clé avec les ABR

Dans une liste de N éléments, une recherche a une complexité de $O(N)$.

Avec les ABR, la complexité devient $O(\text{hauteur de l'arbre}) = O(2^n - 1) = O(N)$

Hauteur minimale d'un arbre pour $N = 2^n - 1$ éléments : $n = O(\ln(N))$.

La complexité de la recherche avec les ABR sera réduite, si la hauteur des arbres est "petite".

```
Recherche : la fonction(donnée x : un entier, donnée A : un AdTriplet) -> un AdTriplet

{Etat initial : x est un entier, A est la racine de l'ABR}
{Etat final : si A contient un noeud de clé x, alors retourne un pointeur vers ce noeud, sinon retourn

Algo :
Si A = Nil, retourne Nil
Sinon
  Si clé de A↑ = x, retourne A
  Sinon
    si clé de A↑ < x, retourne Recherche(x, droit de A↑)
    sinon {clé de A↑ > x}, retourne Recherche(x, gauche de A↑)
```

Insertion dans un ABR

C'est la première fois que l'on verra une action récursive qui modifie un arbre

```
Insérer : l'action(donnée x : un entier, donnée-résultat A : un AdTriplet)

Algo :
Si A = Nil
  Allouer(A)
  Clé de A↑ <- x
  gauche de A↑ <- Nil
  droit de A↑ <- Nil
Sinon
  Si x <= Clé de A↑, Insérer(x, gauche de A↑)
  Sinon Insérer(x, droit de A↑)
```

```
           39
    18           45
12      31          57
  18  27  xx      50  58
```

```
Insérer(34, a)
  Insérer(34, gauche de a↑)
    Insérer(34, droit de b↑)
      Insérer(34, droit de c↑)
```

Le nouveau noeud est une feuille de l'arbre et le chainage est dû à la récursivité.

Suppression d'une clé dans un ABR

```
Supprimer : action(donné x : un entier, donnée-résultat A : un AdTriplet)

Lexique : PMax : des AdTriplet

Algo :

Si A != Nil
  Si x < Clé de A↑ : supprimer(x, gauche de A↑)
  Sinon si > Clé de A↑ : supprimer(x, droit de A↑)
  Sinon {x = Clé de A↑}
    Si gauche de A↑ = Nil et droit de A↑ = Nil : A <- Nil
    Sinon si gauche de A↑ = Nil et droit de A↑ != Nil : A <- droit de A↑
    Sinon si gauche de A↑ != Nil et droit de A↑ = Nil : A <- gauche de A↑
    Sinon {gauche de A↑ != Nil et droit de A↑ != Nil} :
      SupprimerMax(gauche de A↑, PMax)
      gauche de PMax↑ <- gauche de A↑
      droit de PMaX↑ <- droit de A↑
      A <- PMax

SupprimerMax : action(donnée-résultat A, Pmax : des AdTriplets)

Algo :

Si droit de A↑ != Nil, SupprimerMax(droit de A↑, PMax)
Sinon
  PMax <- A
  A <- gauche de A↑

.
```