Programmation fonctionnelle

Cours n°6 : Parcours, arbres binaires de recherche.

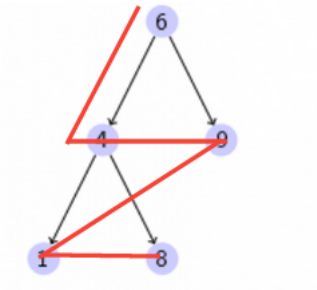
Licence 1 Informatique, Université Paris 8

7 Novembre 2022

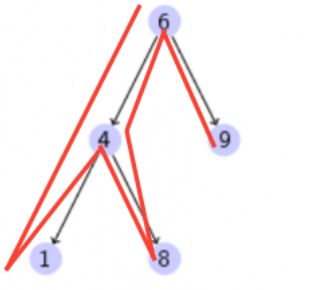
I. Parcours d'arbres

Parcours d'arbre

- Pour pouvoir accéder aux données stockées dans un arbre, il faut **parcourir** cet arbre. Il existe principalement deux types de parcours:
  - Parcours **en largeur**,
  - Parcours **en profondeur**,



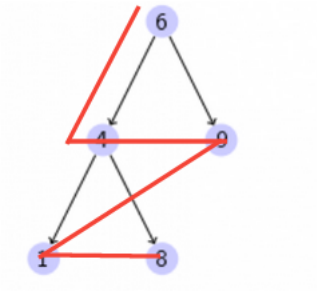
Parcours en largeur



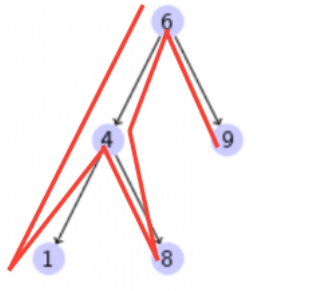
Parcours en profondeur

Parcours d'arbre

- Pour pouvoir accéder aux données stockées dans un arbre, il faut **parcourir** cet arbre. Il existe principalement deux types de parcours:
  - Parcours **en largeur**,
  - Parcours **en profondeur**,



Parcours en largeur



Parcours en profondeur

- **En profondeur** : on passe les nœuds en revue en commençant toujours par le même fils, puis en descendant le plus profondément possible dans l'arbre. Lorsqu'on arrive sur un arbre vide, on remonte jusqu'au noeud supérieur et on redescend dans le fils encore inexploré.
- **En largeur** : on passe en revue étage par étage: d'abord la racine, puis les racines des fils gauche et droit, puis les racines de leurs fils, etc.

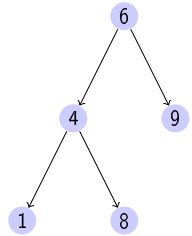
Parcours en profondeur

- Il existe 3 types de parcours en profondeur, dépendant de l'ordre d'affichage des éléments:
  - Parcours **infixe** : Fils gauche, puis racine, puis fils droit.
  - Parcours **préfixe** : Racine, puis fils gauche, puis fils droit.
  - Parcours **postfixe** : Fils gauche, puis fils droit, puis racine.

Parcours en profondeur

- Il existe 3 types de parcours en profondeur, dépendant de l'ordre d'affichage des éléments:
  - Parcours **infixe** : Fils gauche, puis racine, puis fils droit.
  - Parcours **préfixe** : Racine, puis fils gauche, puis fils droit.
  - Parcours **postfixe** : Fils gauche, puis fils droit, puis racine.
- **Exemple** : Pour

$a =$



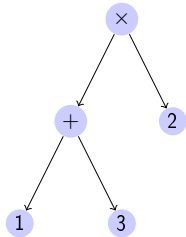
`(infixe a)` renvoie `'(1 4 8 6 9)`.

`(prefixe a)` renvoie `'(6 4 1 8 9)`.

`(postfixe a)` renvoie `'(1 8 4 9 6)`.

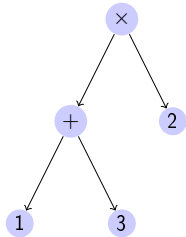
Exemple : calculs avec opérations

- Un calcul arithmétique utilisant les opérations $+, -, \times$ et $\div$ peut être représenté par un arbre, par exemple:



Exemple : calculs avec opérations

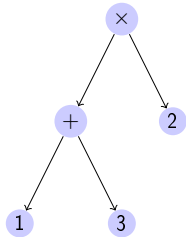
- Un calcul arithmétique utilisant les opérations $+, -, \times$ et $\div$ peut être représenté par un arbre, par exemple:



- Parcours **infixe** : $((1 + 3) \times 2)$. C'est la notation mathématique usuelle.

Exemple : calculs avec opérations

- Un calcul arithmétique utilisant les opérations $+, -, \times$ et $\div$ peut être représenté par un arbre, par exemple:



- Parcours **infixe** : $((1 \ + \ 3) \ \times \ 2)$. C'est la notation mathématique usuelle.

- Parcours **préfixe** : $(\times \ (+ \ 1 \ 3) \ 2)$. C'est la notation de Racket, d'où le nom de **notation préfixée**.

Parcours en profondeur infixe

- **Rappel :** Pour le parcours infixe, d'abord le fils gauche, puis la racine, puis le fils droit.

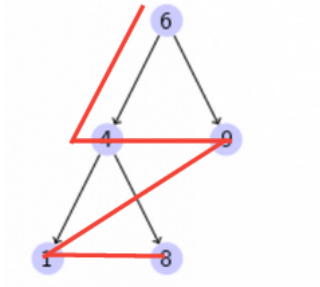
Parcours en profondeur infixe

- **Rappel** : Pour le parcours infixe, d'abord le fils gauche, puis la racine, puis le fils droit.

```
(define (infixe a)
        (if (arbre-vide? a)
            '()
            (append (infixe (fils-g a))
                    (list (racine a))
                    (infixe (fils-d a)))))
```

Parcours en largeur

- On rappelle que le parcours en largeur se fait étage par étage:



- On va donc travailler sur une liste L contenant des arbres, dont on veut chacun récupérer les racines, et les sous-arbres.

Parcours en largeur II

- Récupérer la liste des racines:

```
(define (liste-racines L)
  (if (null? L)
      '()
      (cons (racine (car L)) (liste-racines (cdr L)))))
```

Parcours en largeur II

- Récupérer la liste des racines:

```
(define (liste-racines L)
  (if (null? L)
      '()
      (cons (racine (car L)) (liste-racines (cdr L)))))
```

- Récupérer la liste des sous-arbres non vides:

```
(define (sous-arbres L)
  (cond ((null? L) '())
        ((feuille? (car L)) (sous-arbres (cdr L)))
        ((arbre-vide? (fils-g (car L)))
              (cons (fils-d (car L)) (sous-arbres (cdr L))))
        ((arbre-vide? (fils-d (car L)))
              (cons (fils-g (car L)) (sous-arbres (cdr L))))
        (else
              (cons (fils-g (car L)) (cons (fils-d (car L)) (sous-arbres (cdr L)))))
```

Parcours en largeur II

- Récupérer la liste des racines:

```
(define (liste-racines L)
  (if (null? L)
      '()
      (cons (racine (car L)) (liste-racines (cdr L)))))
```

- Récupérer la liste des sous-arbres non vides:

```
(define (sous-arbres L)
  (cond ((null? L) '())
        ((feuille? (car L)) (sous-arbres (cdr L)))
        ((arbre-vide? (fils-g (car L)))
              (cons (fils-d (car L)) (sous-arbres (cdr L))))
        ((arbre-vide? (fils-d (car L)))
              (cons (fils-g (car L)) (sous-arbres (cdr L))))
        (else
              (cons (fils-g (car L)) (cons (fils-d (car L)) (sous-arbres (cdr L)))))))
```

- Parcours en largeur, avec récursivité terminale

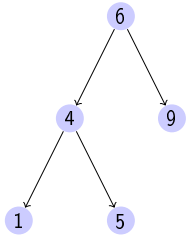
```
(define (largeur a)
  (define (largeur-aux L R)
    (if (null? L)
        R
        (largeur-aux (sous-arbres L) (append R (liste-racines L)))))
  (largeur-aux (list a) '()))
```

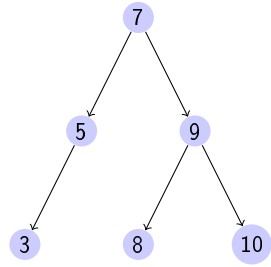
II. Arbres binaires de recherche

Arbres binaires de recherche

- Un **arbre binaire de recherche** (ABR) est un arbre binaire dans lequel les valeurs des nœuds sont ordonnées selon la règle suivante: en chaque nœud, la valeur est
  - (strictement) supérieure à toutes celles de son fils gauche,
  - (strictement) inférieure à toutes celles de son fils droit.

- **Exemples :**





- On suppose qu'un ABR ne contient jamais deux fois la même valeur.

Recherche dans un arbre

- Dans un arbre binaire quelconque, pour tester l'appartenance d'un élément x à un arbre a, on est obligé de comparer x avec toutes les valeurs de l'arbre a jusqu'à ce qu'on le trouve, ou non:

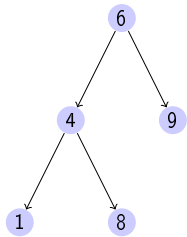
```
(define (appartient x a)
  (if (arbre-vide? a)
      #f
      (or (equal? (racine a) x) (appartient x (fils-g a))
                                (appartient x (fils-d a)))))
```

Recherche dans un arbre

- Dans un arbre binaire quelconque, pour tester l'appartenance d'un élément x à un arbre a, on est obligé de comparer x avec toutes les valeurs de l'arbre a jusqu'à ce qu'on le trouve, ou non:

```
(define (appartient x a)
  (if (arbre-vide? a)
      #f
      (or (equal? (racine a) x) (appartient x (fils-g a))
                                (appartient x (fils-d a)))))
```

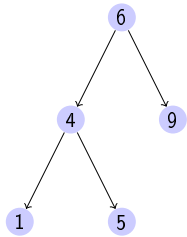
- Si on veut tester si 8 appartient à



on doit tester si il est égal à la racine, et sinon aller chercher si il est dans le fils gauche, ou dans le fils droit. On parcourt ainsi tous les nœuds de l'arbre.

Recherche dans un ABR

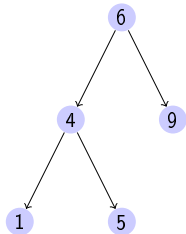
- Les ABR portent bien leur nom: la recherche d'un élément y est simplifiée.
- **Exemple :** Pour trouver 9 dans l'arbre ci-dessous, il suffit de comparer 9 à la racine.



  - si les valeurs sont égales, on renvoie directement $\#t$,
  - si 9 est strictement supérieur, on sait puisque l'arbre est ordonné que 9, si il est dans l'arbre, est forcément dans le fils droit. Il suffit donc d'appeler récursivement dans le fils droit, et on n'aura pas à parcourir le fils gauche.
  - si 9 est strictement inférieur, on appelle récursivement sur le fils gauche et on ne parcourt pas le fils droit.

Recherche dans un ABR

- Les ABR portent bien leur nom: la recherche d'un élément y est simplifiée.
- **Exemple :** Pour trouver 9 dans l'arbre ci-dessous, il suffit de comparer 9 à la racine.

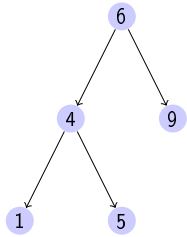


  - si les valeurs sont égales, on renvoie directement $\#t$,
  - si 9 est strictement supérieur, on sait puisque l'arbre est ordonné que 9, si il est dans l'arbre, est forcément dans le fils droit. Il suffit donc d'appeler récursivement dans le fils droit, et on n'aura pas à parcourir le fils gauche.
  - si 9 est strictement inférieur, on appelle récursivement sur le fils gauche et on ne parcourt pas le fils droit.

```
(define (appartient-abr a x)
  (cond ((arbre-vide? a) #f)
        ((equal? (racine a) x) #t)
        ((> (racine a) x) (appartient-abr (fils-g a)))
        (else (appartient-abr (fils-d a)))))
```

Recherche dans un ABR

- Les ABR portent bien leur nom: la recherche d'un élément y est simplifiée.
- **Exemple :** Pour trouver 9 dans l'arbre ci-dessous, il suffit de comparer 9 à la racine.



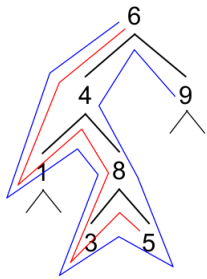
  - si les valeurs sont égales, on renvoie directement $\#t$,
  - si 9 est strictement supérieur, on sait puisque l'arbre est ordonné que 9, si il est dans l'arbre, est forcément dans le fils droit. Il suffit donc d'appeler récursivement dans le fils droit, et on n'aura pas à parcourir le fils gauche.
  - si 9 est strictement inférieur, on appelle récursivement sur le fils gauche et on ne parcourt pas le fils droit.

```
(define (appartient-abr a x)
  (cond ((arbre-vide? a) #f)
        ((equal? (racine a) x) #t)
        ((> (racine a) x) (appartient-abr (fils-g a)))
        (else (appartient-abr (fils-d a)))))
```

- Dans le pire des cas, on parcourt toute une branche, mais pas tous les nœuds !

Résumé

- Dans un arbre binaire quelconque:

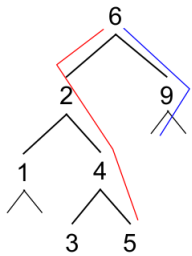


Recherche fructueuse :
Chercher 5

Cas le pire
Recherche infructueuse :
Chercher 7

Complexité au pire :
nombre de nœuds de l'arbre

- Dans un ABR:



Recherche fructueuse :
Chercher 5

Recherche infructueuse :
Chercher 7

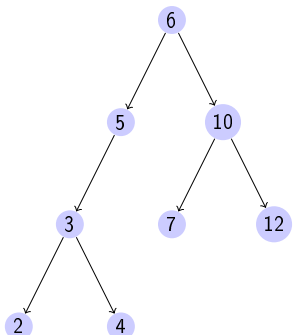
Complexité au pire :
hauteur de l'arbre

Insertion dans un ABR

- Pour insérer dans un arbre binaire, on va toujours insérer aux feuilles, c'est-à-dire en bas de l'arbre.
- On va comparer la valeur à insérer avec la racine: si elle est plus grande, on ira insérer quelque part dans le fils droit; si elle est plus petite, on ira insérer quelque part dans le fils gauche.

Insertion dans un ABR

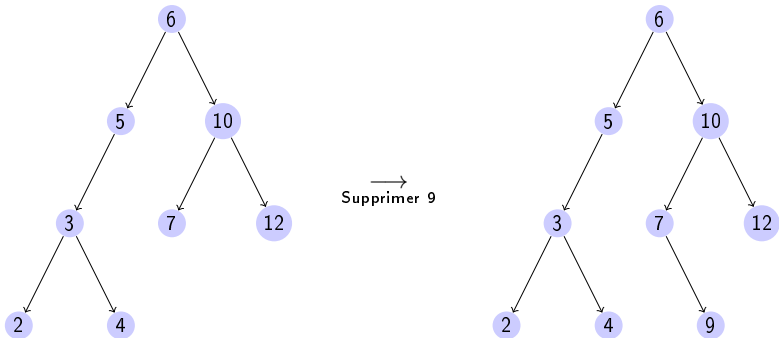
- Pour insérer dans un arbre binaire, on va toujours insérer aux feuilles, c'est-à-dire en bas de l'arbre.
- On va comparer la valeur à insérer avec la racine: si elle est plus grande, on ira insérer quelque part dans le fils droit; si elle est plus petite, on ira insérer quelque part dans le fils gauche.
- **Exemple :** Insérer 9 dans



- On va chercher dans le fils droit car $9 > 6$, puis dans le fils gauche car $9 < 10$, puis puisque $9 > 7$ on insère 9 en fils droit de la feuille 7.

Insertion dans un ABR

- Pour insérer dans un arbre binaire, on va toujours insérer aux feuilles, c'est-à-dire en bas de l'arbre.
- On va comparer la valeur à insérer avec la racine: si elle est plus grande, on ira insérer quelque part dans le fils droit; si elle est plus petite, on ira insérer quelque part dans le fils gauche.
- **Exemple :** Insérer 9 dans



- On va chercher dans le fils droit car $9 > 6$, puis dans le fils gauche car $9 < 10$, puis puisque $9 > 7$ on insère 9 en fils droit de la feuille 7.

Suppression dans un ABR

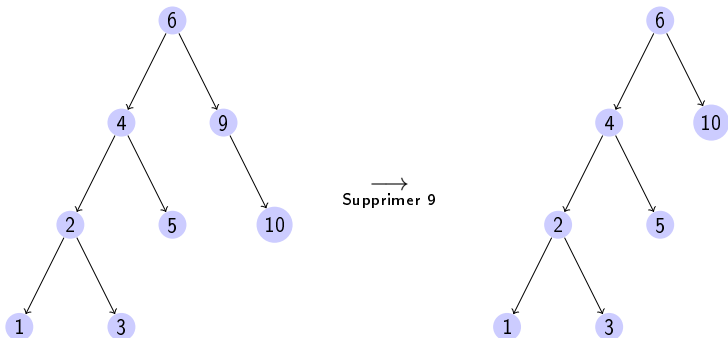
- On veut être capable de supprimer un élément x d'un ABR a, tout en s'assurant qu'on garde bien la condition d'être un ABR.

Suppression dans un ABR

- On veut être capable de supprimer un élément x d'un ABR a, tout en s'assurant qu'on garde bien la condition d'être un ABR.
- Si le nœud qu'on souhaite supprimer est une feuille, on peut le supprimer sans craindre de perdre la structure d'ABR.

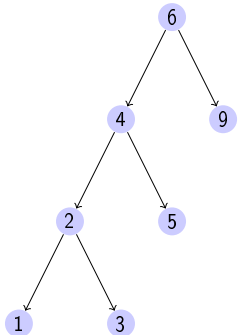
Suppression dans un ABR

- On veut être capable de supprimer un élément x d'un ABR a, tout en s'assurant qu'on garde bien la condition d'être un ABR.
- Si le nœud qu'on souhaite supprimer est une feuille, on peut le supprimer sans craindre de perdre la structure d'ABR.
- Si le nœud qu'on souhaite supprimer admet un seul fils qui est non vide, on supprime le nœud et on décale le fils d'un cran vers le haut.



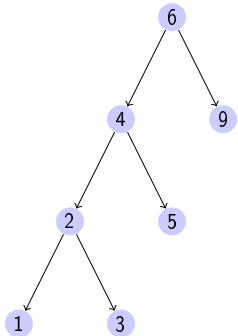
Suppression dans un ABR II

- Le cas le plus compliqué est de supprimer un élément qui a deux fils, par exemple pour supprimer 4 ci-dessous.
- **Exemple :**



Suppression dans un ABR II

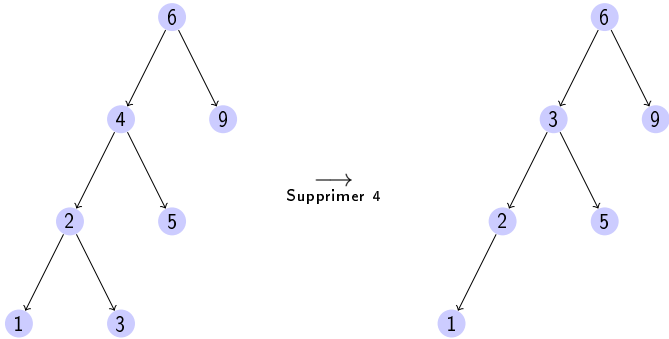
- Le cas le plus compliqué est de supprimer un élément qui a deux fils, par exemple pour supprimer 4 ci-dessous.
- **Exemple :**



- Pour supprimer 4, on doit mettre à sa place un nœud qui est compris entre la racine de son fils gauche (2) et la racine de son fils droit (5).
- Pour trouver un tel nœud, on peut par exemple choisir le maximum du fils gauche de 4, ici 3 ou le minimum du fils droit de 4, ici 5. **On va donc supprimer selon une stratégie.**

Suppression dans un ABR II

- Le cas le plus compliqué est de supprimer un élément qui a deux fils, par exemple pour supprimer 4 ci-dessous.
- **Exemple :**



- Pour supprimer 4, on doit mettre à sa place un nœud qui est compris entre la racine de son fils gauche (2) et la racine de son fils droit (5).
- Pour trouver un tel nœud, on peut par exemple choisir le maximum du fils gauche de 4, ici 3 ou le minimum du fils droit de 4, ici 5. **On va donc supprimer selon une stratégie.**

Suppression dans un ABR III

- En général, on supprimera selon une stratégie:
  - soit on supprime le nœud et on le remplace par le maximum du fils gauche, et on supprime ce dernier,
  - soit on supprime le nœud et on le remplace par le minimum du fils droit, et on supprime ce dernier.

Suppression dans un ABR III

- En général, on supprimera selon une stratégie:
  - soit on supprime le nœud et on le remplace par le maximum du fils gauche, et on supprime ce dernier,
  - soit on supprime le nœud et on le remplace par le minimum du fils droit, et on supprime ce dernier.
- La suppression du minimum ou du maximum est facile à gérer puisque par propriété d'un ABR, on sera nécessairement dans un des cas feuille ou nœud à un seul fils.
- Voici ci-dessous l'idée de l'algorithme:
  - Si a est un arbre vide, on renvoie un arbre vide;
  - Si $x < \text{racine}(a)$, on renvoie un arbre constitué de la racine de a, le fils gauche de a dans lequel on supprime x récursivement, et le fils droit de a.
  - Si $x > \text{racine}(a)$, on renvoie un arbre constitué de la racine de a, le fils gauche de a, et le fils droit de a dans lequel on supprime x récursivement.
  - Si $x = \text{racine}(a)$, alors si x est une feuille on retourne un arbre vide. Si x a un seul fils, on supprime x et on recolle l'unique fils de x à la place. Si x a deux fils, on remplace x par le maximum du fils gauche de x, et on supprime ce maximum.