Algorithmique des structures arborescentes
Algorithmique et programmation fonctionnelle

L2 Info, Math-info & CMI ISI, 2022–2023

Marc Zeitoun

19 janvier 2023

Plan

La structure d'arbre

Arbres binaires : vocabulaire & propriétés

Arbres binaires de recherche

Comparaison C vs. OCaml (démo)

La structure d'arbre

Structure de données

Une structure de donnée est

Structure de données

Une structure de donnée est une **façon d'organiser des données**.

Structure de données

Une structure de donnée est une **façon d'organiser des données**.

Structures de données déjà rencontrées :

Structure de données

Une structure de donnée est une **façon d'organiser des données**.

Structures de données déjà rencontrées :

- Tableaux,
- Listes,
- Piles,
- Files,
- Tables de hachage.

Structure de données

Une structure de donnée est une **façon d'organiser des données**.

Structures de données déjà rencontrées :

- Tableaux,
- Listes,
- Piles,
- Files,
- Tables de hachage.

Objectif d'une structure de données :

Structure de données

Une structure de donnée est une **façon d'organiser des données**.

Structures de données déjà rencontrées :

- Tableaux,
- Listes,
- Piles,
- Files,
- Tables de hachage.

Objectif d'une structure de données :
Permettre un accès et une modification **rapide** des données.

Opérations habituelles sur une structure de données

Opérations habituelles sur une structure de données

- Chercher une donnée,

Opérations habituelles sur une structure de données

- Chercher une donnée,
- Ajouter une donnée,

Opérations habituelles sur une structure de données

- Chercher une donnée,
- Ajouter une donnée,
- Supprimer une donnée,

Opérations habituelles sur une structure de données

- Chercher une donnée,
- Ajouter une donnée,
- Supprimer une donnée,
- Trouver la plus petite donnée,

Opérations habituelles sur une structure de données

- Chercher une donnée,
- Ajouter une donnée,
- Supprimer une donnée,
- Trouver la plus petite donnée,
- Lister toutes les données dans l'ordre.
- ...

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau				

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Tableau trié				

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Tableau trié	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Tableau trié	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Liste				

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Tableau trié	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Liste	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Tableau trié	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Liste	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Arbres AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Complexité des opérations selon les structures

$n =$ nombre d'éléments stockés dans la structure.

	Recherche	Insertion	Suppression	Recherche minimum
Tableau	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Tableau trié	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$
Liste	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Arbres AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Tas binaire	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$
			Suppression mininum	
			$O(\log n)$	

À retenir

- **Structure de données** : organisation d'éléments en mémoire.

À retenir

- **Structure de données** : organisation d'éléments en mémoire.
- **Qualité** d'une structure de données :
  - Simplicité de conception, de programmation.
  - Coût des opérations.

À retenir

- **Structure de données** : organisation d'éléments en mémoire.
- **Qualité** d'une structure de données :
  - Simplicité de conception, de programmation.
  - Coût des opérations.
- Pas de structure ultime : performances souvent incomparables.

$\Longrightarrow$ choix de structure en fonction de ce qu'on souhaite faire.

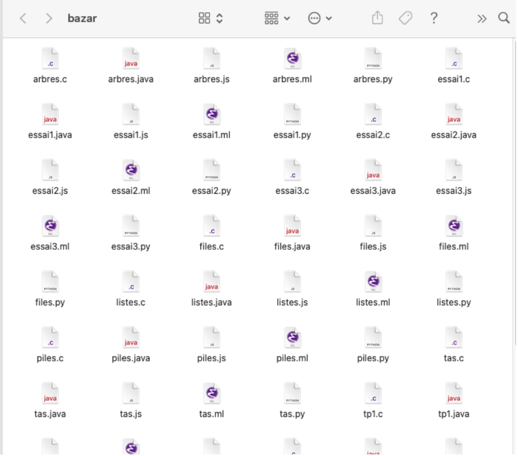
Les arbres

Arbres : un structures de données récursive

Dans la vie courante ?

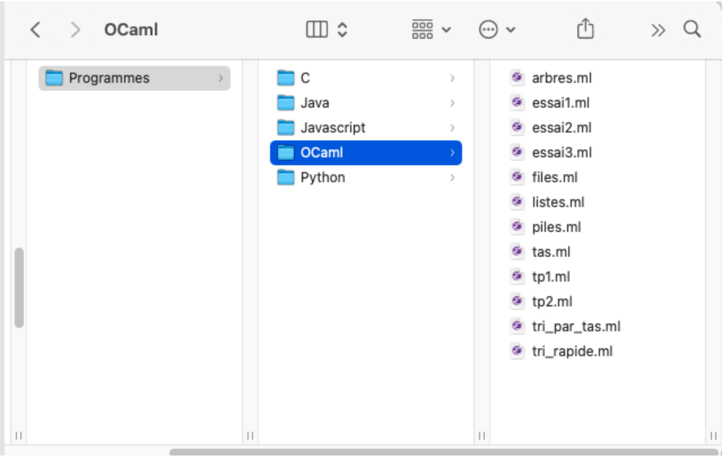
Arbres : un structures de données récursive

Dans la vie courante ?



Arbres : un structures de données récursive

Dans la vie courante ?



La structure d'arbre

Arbres binaires : vocabulaire & propriétés

- Vocabulaire sur les arbres
- Propriétés des arbres binaires
  - Comment rédiger une preuve ?

Arbres binaires de recherche

- Recherche, Maximum, Insertion, Suppression

Comparaison C vs. OCaml (démo)

- Parcours en profondeur

L'arbre comme structure de données

- Apparaît naturellement : HTML, arborescences de répertoires, ...
- Structure qui permet de ranger efficacement des données.
- Gain de temps spectaculaire par rapport à listes/piles/files.
  - **10000 ans** → quelques secondes.
- Prérequis pour algo des graphes.

Il y a plusieurs sortes d'arbres. Aujourd'hui : arbres **binaires**.

Arbres binaires : définition

Définition récursive. Un ***arbre binaire étiqueté*** est :

- soit l'arbre vide, noté $()$ ou Empty dans ce cours.
- soit est formé
  - d'un nœud, appelé sa **racine**, portant une information appelée **étiquette**, ou **clé** du nœud,
  - et de deux arbres binaires, appelés **sous-arbres** gauche et droit.

Arbres binaires : définition

Définition récursive. Un ***arbre binaire étiqueté*** est :

- soit l'arbre vide, noté $()$ ou Empty dans ce cours.
- soit est formé
  - d'un nœud, appelé sa **racine**, portant une information appelée **étiquette**, ou **clé** du nœud,
  - et de deux arbres binaires, appelés **sous-arbres** gauche et droit.

Un arbre non vide t est donc décrit par un triplet (v, ℓ, r) formé :

- d'une valeur v de type fixé, qui est l'étiquette de la racine de t,
- d'un arbre ℓ, qui est le sous-arbre gauche de t,
- d'un arbre r, qui est le sous-arbre droit de t.

Arbres binaires : représentation graphique

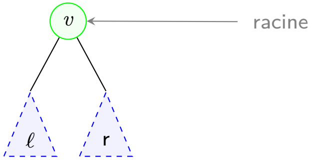
Définition récursive $\Longrightarrow$ **dessin** obtenu récursivement.

- arbre vide : on ne dessine rien.

Arbres binaires : représentation graphique

Définition récursive $\Longrightarrow$ **dessin** obtenu récursivement.

- arbre vide : on ne dessine rien.
- arbre non vide (v, ℓ, r) :

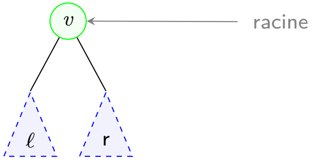


et le dessin continue, récursivement, pour ℓ et r.

Arbres binaires : représentation graphique

Définition récursive $\Longrightarrow$ **dessin** obtenu récursivement.

- arbre vide : on ne dessine rien.
- arbre non vide (v, ℓ, r) :



et le dessin continue, récursivement, pour ℓ et r.

Attention !

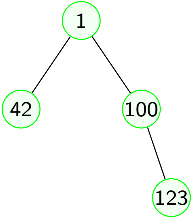
1. L'arbre vide n'est pas un nœud !
2. Bien distinguer : arbre, nœud, valeur (types différents !).

Exemple

$$t = (\underbrace{1}_{v}, \quad \underbrace{(42, (), ())}_{\ell}, \quad \underbrace{(100, (), (123, (), ()))}_{r})$$

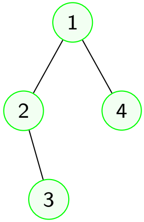
Exemple

$$t = (\underbrace{1}_{v}, \quad \underbrace{(42, (), ())}_{\ell}, \quad \underbrace{(100, (), (123, (), ()))}_{r})$$

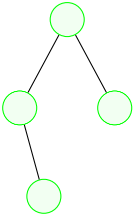


Squelette d'un arbre

Squelette d'un arbre binaire : obtenu en supprimant les étiquettes.



(a) Un arbre binaire



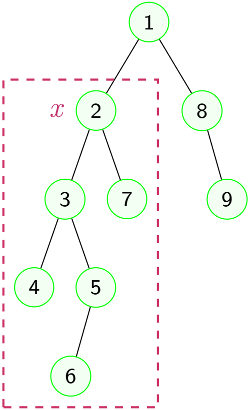
(b) Son squelette

Vocabulaire à connaître

- Sous-arbre enraciné en un nœud,
- Fils gauche, fils droit d'un nœud,
- Père d'un nœud,
- Arité d'un nœud,
- Feuille,
- Arête,
- Branche,
- Hauteur d'un arbre,
- Taille d'un arbre,
- Profondeur (ou niveau) d'un nœud.

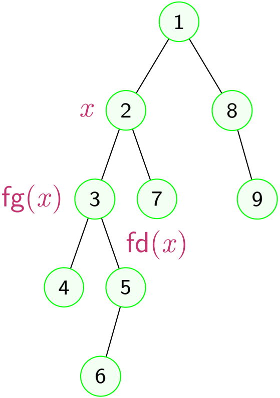
Sous-arbre enraciné en un nœud x

Arbre situé « en dessous du nœud x » (x est inclus).



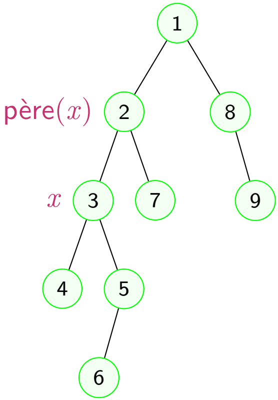
Fils gauche, fils droit d'un nœud x

Fils gauche du nœud x, noté $\text{fg}(x) =$ racine du sous-arbre gauche.



- Le nœud d'étiquette 5 a un fils gauche mais pas de fils droit,
- Celui d'étiquette 8 a un fils droit mais pas de fils gauche,
- Ceux d'étiquettes 4, 6, 7, 9 n'ont pas de fils.

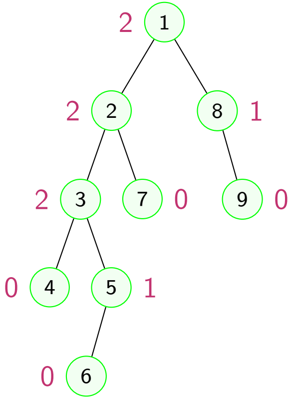
Père d'un nœud



Tout nœud sauf la racine a un (seul) père.

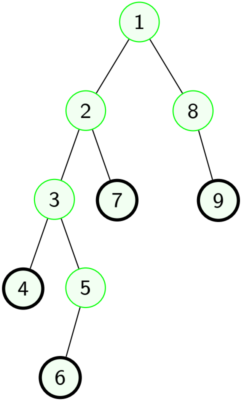
Arité d'un nœud

Arité d'un nœud x = nombre de fils de x.



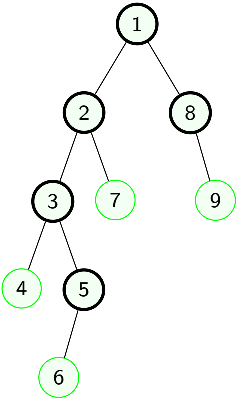
Feuille

Feuille = nœud dont l'arité est 0.



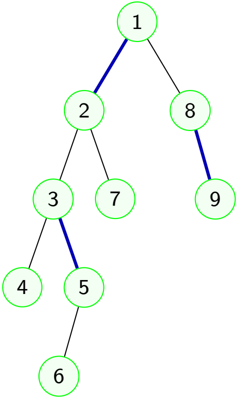
Nœud interne

Nœud interne = nœud qui n'est pas une feuille.



Arête

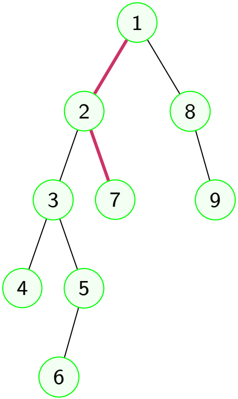
Relie un fils et son père.



Cet arbre a 8 arêtes (3 sont en bleu).

Branche

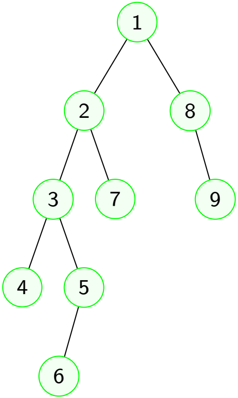
Chemin de la racine à une feuille.



- Exemple : $1 \to 2 \to 7$.
- Une branche descend de la racine jusqu'à une feuille.
- Longueur d'une branche = son nombre d'arêtes.

Taille d'un arbre

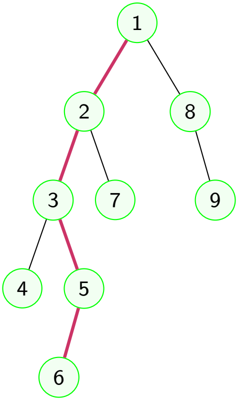
Nombre de nœuds.



Cet arbre a comme taille 9.

Hauteur d'un arbre

Longueur maximale d'une branche.

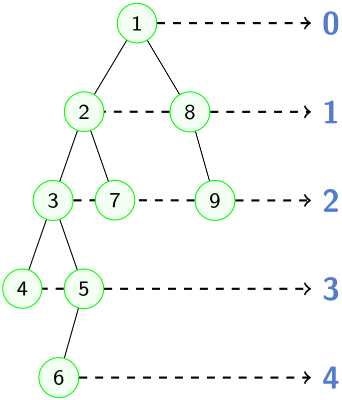


La hauteur de cet arbre est 4.

Par convention, la hauteur de l'arbre vide est -1.

Profondeur (ou niveau) d'un nœud

Nombre d'arêtes qui séparent le nœud de la racine.



Le terme **niveau** est synonyme de **profondeur**.

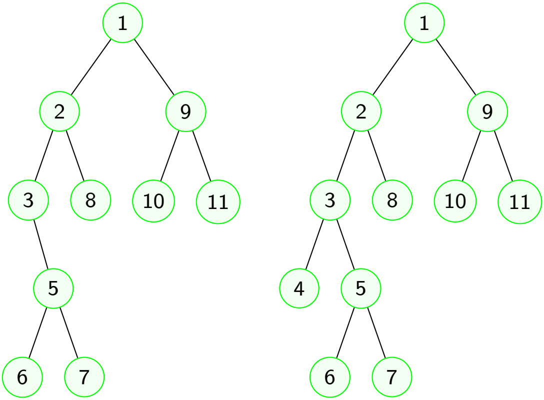
Arbres binaires particuliers

Un arbre binaire est

- complet s'il n'a pas de nœud d'arité 1.
- parfait s'il est complet et toutes les feuilles ont même niveau.
- quasi-parfait si
  - il est complet jusqu'au niveau $h-1$, et
  - ses feuilles sont à profondeur h ou $h-1$, et
  - les feuilles de profondeur h sont « le plus à gauche possible ».

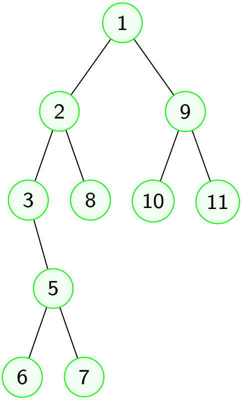
  Remarque : tout arbre parfait est quasi-parfait.

Arbre complet

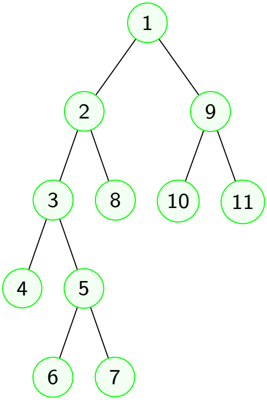


Arbre complet

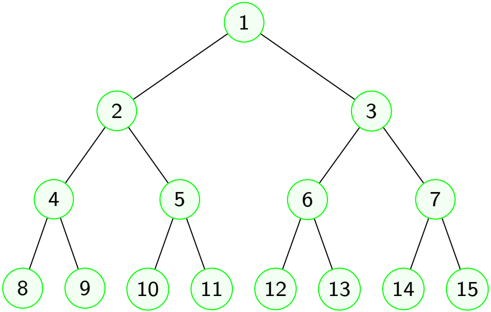
Non complet



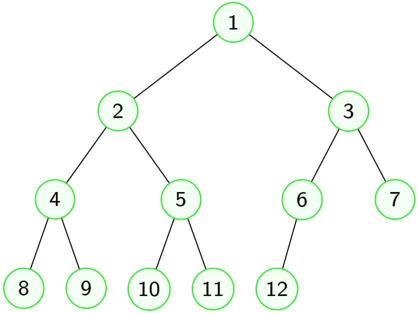
Complet



Arbre parfait



Arbre quasi-parfait



Propriétés sur les arbres

- La hauteur, nombre de nœuds, nombre de feuilles, etc., vérifient des relations simples.
- Un **objectif** de l'UE est que vous devez comprendre comment montrer ces relations par récurrence.
- Récurrences : sur la taille ou la hauteur des arbres.

Comment rédiger une preuve ?

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓
- Supposons le résultat vrai pour tout arbre non vide de taille $\leqslant n - 1$.

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓
- Supposons le résultat vrai pour tout arbre non vide de taille $\leqslant n - 1$.
- Soit t de taille $n > 1$.
  - Comme t est formé d'un nœud racine et de sous-arbres ℓ et r :

$$n = 1 + s(\ell) + s(r). \tag{1}$$

    où $s(\ell) =$ taille de ℓ et $s(r) =$ taille de r.

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓
- Supposons le résultat vrai pour tout arbre non vide de taille $\leqslant n - 1$.
- Soit t de taille $n > 1$.
  - Comme t est formé d'un nœud racine et de sous-arbres ℓ et r :
    $$n = 1 + s(\ell) + s(r). \tag{1}$$
    où $s(\ell) =$ taille de ℓ et $s(r) =$ taille de r.
  - Comme $n > 1$, soit ℓ soit r n'est pas vide, par exemple ℓ.

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓
- Supposons le résultat vrai pour tout arbre non vide de taille $\leqslant n - 1$.
- Soit t de taille $n > 1$.
  - Comme t est formé d'un nœud racine et de sous-arbres ℓ et r :
  $$n = 1 + s(\ell) + s(r). \tag{1}$$
  où $s(\ell) =$ taille de ℓ et $s(r) =$ taille de r.
  - Comme $n > 1$, soit ℓ soit r n'est pas vide, par exemple ℓ.
  - D'après l'équation (1), on a $s(\ell) \leqslant n - 1$.

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓
- Supposons le résultat vrai pour tout arbre non vide de taille $\leqslant n-1$.
- Soit t de taille $n > 1$.
  - Comme t est formé d'un nœud racine et de sous-arbres ℓ et r :

    $$n = 1 + s(\ell) + s(r). \tag{1}$$

    où $s(\ell) =$ taille de ℓ et $s(r) =$ taille de r.
  - Comme $n > 1$, soit ℓ soit r n'est pas vide, par exemple ℓ.
  - D'après l'équation (1), on a $s(\ell) \leqslant n-1$.
  - Par hypothèse de récurrence, ℓ a au moins une feuille.

Tout arbre binaire non vide a au moins une feuille

On raisonne par récurrence sur la taille $s(t)$ de l'arbre t.

- Comme t est non vide, on a $s(t) \geqslant 1$.
- Si $s(t) = 1$, alors la racine de t est une feuille. ✓
- Supposons le résultat vrai pour tout arbre non vide de taille $\leqslant n - 1$.
- Soit t de taille $n > 1$.
  - Comme t est formé d'un nœud racine et de sous-arbres ℓ et r :
    $$n = 1 + s(\ell) + s(r). \tag{1}$$
    où $s(\ell) =$ taille de ℓ et $s(r) =$ taille de r.
  - Comme $n > 1$, soit ℓ soit r n'est pas vide, par exemple ℓ.
  - D'après l'équation (1), on a $s(\ell) \leqslant n - 1$.
  - Par hypothèse de récurrence, ℓ a au moins une feuille.
  - Cette feuille est aussi une feuille de t. ✓

Propriétés des arbres binaires

Si t est un arbre, on note

- $h(t)$ sa hauteur,
- $s(t)$ son nombre de nœuds.
- $i(t)$ son nombre de nœuds internes.
- $\ell(t)$ son nombre de feuilles.

Comment calculer récursivement ces paramètres ?

Propriétés (se montrent par récurrence)

1. On a $\leqslant s(t) \leqslant$.

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant$.

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si

$$h(t) + 1 = s(t).$$

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si

$$h(t) + 1 = s(t).$$

3. L'arbre t est parfait si et seulement si

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si
$$h(t) + 1 = s(t).$$
3. L'arbre t est parfait si et seulement si
$$s(t) = 2^{h(t)+1} - 1.$$

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si
$$h(t) + 1 = s(t).$$
3. L'arbre t est parfait si et seulement si
$$s(t) = 2^{h(t)+1} - 1.$$
4. Si t est un arbre **complet**, on a $i(t) =$

Propriétés (se montrent par récurrence)

1. On a $h(t) + 1 \leqslant s(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si
$$h(t) + 1 = s(t).$$
3. L'arbre t est parfait si et seulement si
$$s(t) = 2^{h(t)+1} - 1.$$
4. Si t est un arbre **complet**, on a $i(t) = \ell(t) - 1$.

La structure d'arbre

Arbres binaires : vocabulaire & propriétés

- Vocabulaire sur les arbres
- Propriétés des arbres binaires
  - Comment rédiger une preuve ?

Arbres binaires de recherche

- Recherche, Maximum, Insertion, Suppression

Comparaison C vs. OCaml (démo)

- Parcours en profondeur

Les arbres binaires de recherche (ABR)

- Arbre dans lesquels les clés sont triées.

Les arbres binaires de recherche (ABR)

- Arbre dans lesquels les clés sont triées.
- En **tout** nœud x :
  - si y est dans le sous-arbre **gauche** de x :

    $$\text{clé}(y) \leqslant \text{clé}(x)$$

  - si y est dans le sous-arbre **droit** de x :

    $$\text{clé}(y) \geqslant \text{clé}(x)$$

Un tel arbre est appelé arbre binaire de recherche **(ABR)**,
En anglais : binary search tree **(BST)**.

Les arbres binaires de recherche (ABR)

- Arbre dans lesquels les clés sont triées.
- En **tout** nœud x :
  - si y est dans le sous-arbre **gauche** de x :

$$\text{clé}(y) \leqslant \text{clé}(x)$$

  - si y est dans le sous-arbre **droit** de x :

$$\text{clé}(y) \geqslant \text{clé}(x)$$

Un tel arbre est appelé arbre binaire de recherche **(ABR)**,
En anglais : binary search tree **(BST)**.

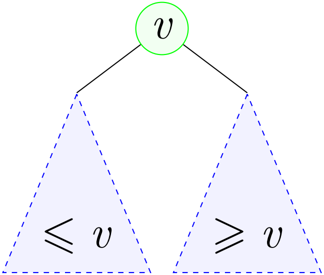
⚠ Cette propriété ne se vérifie pas uniquement « localement ».

ABR

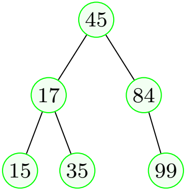
La propriété d'être un arbre binaire de recherche n'est pas locale.

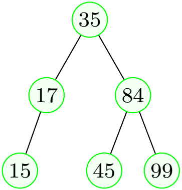
Pour **chaque nœud** v, elle implique :

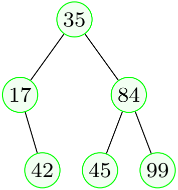
- tous les nœuds du sous-arbre gauche de v,
- tous les nœuds du sous-arbre droit de v.

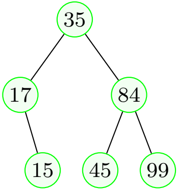


ABR : exemples et non-exemples

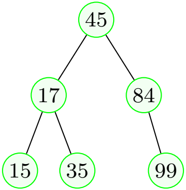




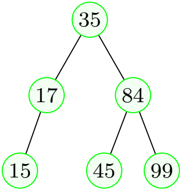


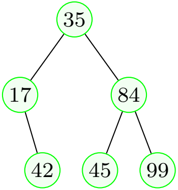


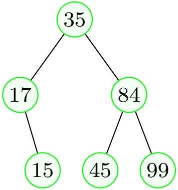
ABR : exemples et non-exemples



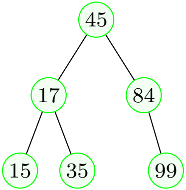
OUI



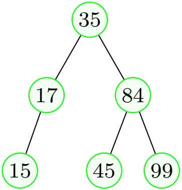




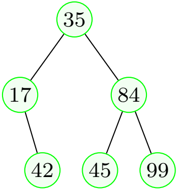
ABR : exemples et non-exemples

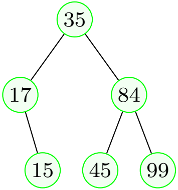


OUI

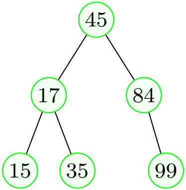


OUI

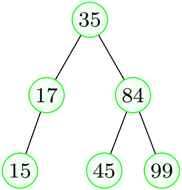




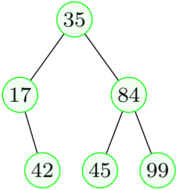
ABR : exemples et non-exemples



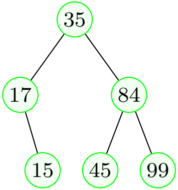
OUI



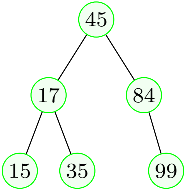
OUI



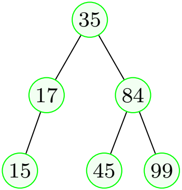
NON



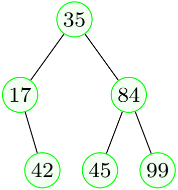
ABR : exemples et non-exemples



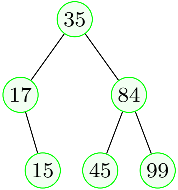
OUI



OUI



NON



NON

Objectif

Manipuler des arbres représentant de grands ensembles qui évoluent.

On veut pouvoir, rapidement :

- chercher un élément particulier, le min, le max,
- ajouter un élément,
- supprimer un élément.

Recherche d'un élément x

Utilise la propriété d'être un arbre binaire ***de recherche***.

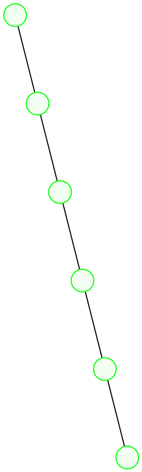
1. Si l'arbre est vide : x n'est **pas présent** dans l'arbre.
2. Si $x =$ clé(racine) : **trouvé**.
3. Si $x <$ clé(racine) : recherche récursive, sous-arbre gauche.
4. Si $x >$ clé(racine) : recherche récursive, sous-arbre droit.

Recherche d'un élément x

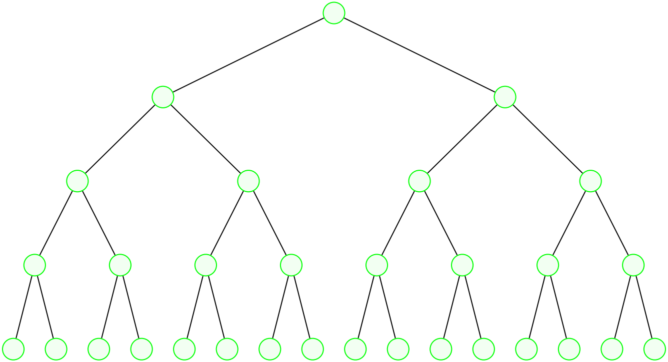
```
type 'a tree = | Empty
               | Bin of 'a * 'a tree * 'a tree

let rec recherche x t =
  match t with
  | Empty -> false
  | Bin(y, l, r) -> (x = y)                   ||
                    (x < y && recherche x l)  ||
                    (x > y && recherche x r)
```

Complexité de la recherche ?

Complexité de la recherche ?



Complexité de la recherche ?



Complexité de la recherche ?

La recherche d'un élément est en $O(h)$, où $h =$ hauteur de l'arbre.
Or, si n est son nombre de nœuds :

$$h + 1 \leqslant n \leqslant 2^{h+1} - 1,$$

Donc

$$\log_2(n+1) - 1 \leqslant h \leqslant n - 1.$$

But : travailler seulement avec des ABR de hauteur $O(\log n)$.
$\rightsquigarrow$ vu plus tard avec des arbres ABR particuliers, les AVL.

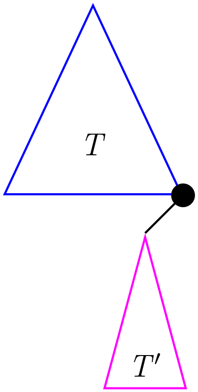
Élément maximum dans un ABR non vide

Où le trouver ?

Élément maximum dans un ABR non vide

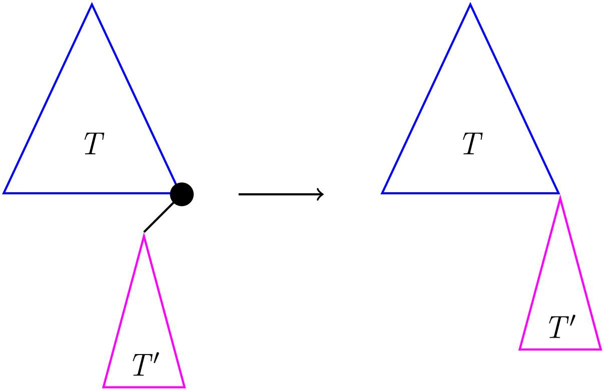
Où le trouver ?

- Se trouve en parcourant l'arbre depuis la racine vers la droite.
- Le nœud portant la clé maximale n'a pas de fils droit.



Suppression de l'élément maximum dans un ABR

Suppression $O(1)$ une fois le nœud ayant la clé maximale localisé.



Suppression de l'élément maximum dans un ABR

Implémentation possible OCaml :

```ocaml
type 'a tree = | Empty
               | Bin of 'a * 'a tree * 'a tree

let rec delete_max t =
  match t with
  | Empty -> failwith "Suppression dans arbre vide"
  | Bin(x, l, r) -> if r = Empty
                    then l
                    else Bin(x, l, delete_max r)
```

Suppression de l'élément maximum dans un ABR

```
let rec delete_max t =
  match t with
  | Empty -> failwith "Suppression dans arbre vide"
  | Bin(x, l, r) -> if r = Empty
                    then l
                    else Bin(x, l, delete_max r)
let e = Empty (* affichage plus compact *)
```

Suppression de l'élément maximum dans un ABR

```
let rec delete_max t =
  match t with
  | Empty -> failwith "Suppression dans arbre vide"
  | Bin(x, l, r) -> if r = Empty
                    then l
                    else Bin(x, l, delete_max r)
let e = Empty (* affichage plus compact *)
```

- `delete_max (Bin(1, Bin(0,e,e), Bin(3, Bin(2,e,e), e)))` entre dans le cas `else` et renvoie :

`Bin(1, Bin(0,e,e), delete_max (Bin(3, Bin(2,e,e), e)))`

Suppression de l'élément maximum dans un ABR

```
let rec delete_max t =
  match t with
  | Empty -> failwith "Suppression dans arbre vide"
  | Bin(x, l, r) -> if r = Empty
                    then l
                    else Bin(x, l, delete_max r)
let e = Empty (* affichage plus compact *)
```

- `delete_max (Bin(1, Bin(0,e,e), Bin(3, Bin(2,e,e), e)))` entre dans le cas `else` et renvoie : `Bin(1, Bin(0,e,e), delete_max (Bin(3, Bin(2,e,e), e)))`
- Le **second** appel `delete_max (Bin(3, Bin(2,e,e), e)))` entre dans le cas `then` et renvoie `Bin(2,e,e)`.

Suppression de l'élément maximum dans un ABR

```
let rec delete_max t =
  match t with
  | Empty -> failwith "Suppression dans arbre vide"
  | Bin(x, l, r) -> if r = Empty
                    then l
                    else Bin(x, l, delete_max r)
let e = Empty (* affichage plus compact *)
```

- `delete_max (Bin(1, Bin(0,e,e), Bin(3, Bin(2,e,e), e)))` entre dans le cas `else` et renvoie : `Bin(1, Bin(0,e,e), delete_max (Bin(3, Bin(2,e,e), e)))`

- Le **second** appel `delete_max (Bin(3, Bin(2,e,e), e)))` entre dans le cas `then` et renvoie `Bin(2,e,e)`.

Le résultat est donc `Bin(1, Bin(0,e,e), Bin(2,e,e))`.

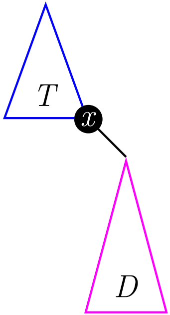
Insertion d'un élément

Principe.

- Naviguer jusqu'à arriver à un sous-arbre vide, où la donnée peut être insérée sans casser la propriété d'être ABR,
- Créer une feuille à cet emplacement.

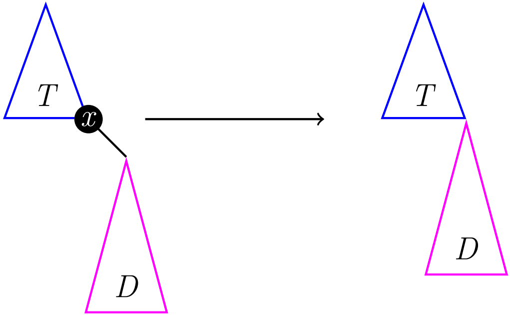
Suppression d'un élément x

- Rechercher le nœud portant la clé x à supprimer.
- 1er cas : son sous-arbre gauche est vide :



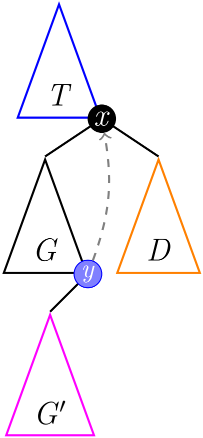
Suppression d'un élément x

- Rechercher le nœud portant la clé x à supprimer.
- 1$^{\text{er}}$ cas : son sous-arbre gauche est vide :



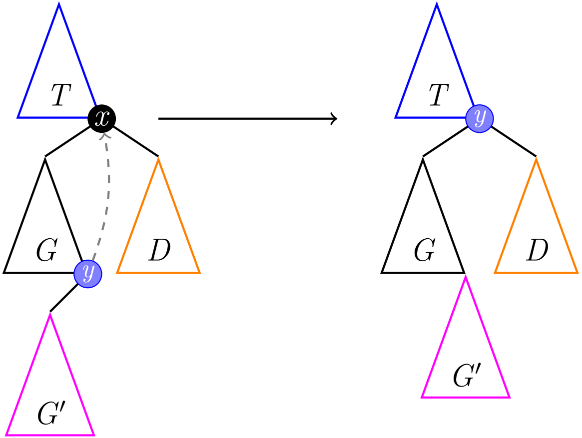
Suppression d'un élément x

- 2e cas : son sous-arbre gauche est non vide : échange avec le maximum y de son sous-arbre gauche, facile à supprimer. Conserve la propriété ABR.



Suppression d'un élément x

- 2^e cas : son sous-arbre gauche est non vide : échange avec le maximum y de son sous-arbre gauche, facile à supprimer. Conserve la propriété ABR.



Parcours en profondeur

- But : effectuer un traitement sur tous les nœuds d'un arbre.
- Se programme très simplement de façon récursive.
- Postfixe :
  - Parcourir le sous-arbre gauche,
  - Parcourir le sous-arbre droit,
  - Effectuer le traitement sur la racine.
- Infixe.
- Préfixe.

Parcours prefixe, infixe, postfixe : exemple

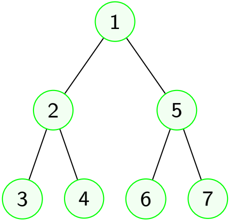


Figure – P2 : Arbre parfait de hauteur 2

Dans quel ordre les nœuds sont-ils traités ?

Parcours prefixe, infixe, postfixe : exemple

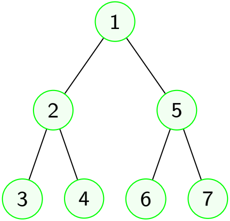


Figure – P2 : Arbre parfait de hauteur 2

Dans quel ordre les nœuds sont-ils traités ?

- Préfixe : 1 2 3 4 5 6 7
- Infixe : 3 2 4 1 6 5 7
- Postfixe : 3 4 2 6 7 5 1

Pourquoi OCaml ?

Bien adapté à la récursion, et les arbres sont une structure récursive.

 Pas besoin d'allouer ni de désallouer.

 Démo !