Algorithmique des structures arborescentes
Algorithmique et programmation fonctionnelle

L2 Info et Math-info, 2019–2020

Marc Zeitoun

14 janvier 2020

Plan

Objectifs et organisation des UE

Arbres binaires : vocabulaire & propriétés

Comparaison C vs. OCaml (démo)

Objectifs et organisation des UE

Objectifs de l'UE

- **Comprendre** les arbres comme structures de données.
- Concept centraux
  - **Récursivité**.
  - **Correction et efficacité des algorithmes**.

Objectifs de l'UE

- **Comprendre** les arbres comme structures de données.
- Concept centraux
  - **Récursivité**.
  - **Correction et efficacité des algorithmes**.

Points importants :

- Comprendre les algorithmes du cours.
- Concevoir des algorithmes simples.
- Évaluer la complexité d'algorithmes simples.
- Montrer des propriétés simples des arbres et algorithmes.
- OCaml pour tests rapides (**pas** pour devenir expert OCaml).

Organisation

- L'enseignement dure **12 semaines**.
- **CM** : **6** $\times$ 1h20.
  Créneau habituel : jeudi 9h30–10h50.

Organisation

- L'enseignement dure **12 semaines**.
- **CM** : **6** $\times$ 1h20.
  Créneau habituel : jeudi 9h30–10h50.
- **CI** et **TM** : 4 blocs de 3 semaines :
  - **2** $\times$ 1h20 CI,
  - 1h20 CI + 1h20 TM.
  - 1h20 CI + 1h20 TM.

Organisation

- L'enseignement dure **12 semaines**.
- **CM** : **6** $\times$ 1h20.
  Créneau habituel : jeudi 9h30–10h50.
- **CI** et **TM** : 4 blocs de 3 semaines :
  - **2** $\times$ 1h20 CI,
  - 1h20 CI + 1h20 TM.
  - 1h20 CI + 1h20 TM.
- Nous joindre : uf-info.ue.algo-arbres@diff.u-bordeaux.fr.

Supports d'enseignement

Le cours est fait en CM **et en CI**.

- Site **Moodle** de l'UE, ouverture prochainement.
  - **Polycopié** de cours à lire **pendant** le semestre.
  - **Diaporamas** de cours.
  - Feuilles d'**exercices**.
  - **Exercices** Moodle.
  - 2 **livres** en français, pdf accessibles gratuitement.

Évaluation

- **Contrôle continu (CC)**.
  - **Tests**.
  - **DS** 1h20, mars ou avril.
  - éventuellement **TP noté**.
  - Évaluation **Moodle**.

Évaluation

- **Contrôle continu (CC)**.
  - **Tests**.
  - **DS** 1h20, mars ou avril.
  - éventuellement **TP noté**.
  - Évaluation **Moodle**.
- **Examen session 1 (EX1) et 2 (EX2)**
- **Note session 1** : (EX1+CC)/2
- **Note session 2** : max(EX2, (EX2+CC)/2)

Comment réussir

- Lire et **comprendre** le cours
- Être **actif/active** en CI et TM
- **Chercher** les exercices soi-même
- Ne pas hésiter à nous contacter par mail

Contenu de l'UE

- Rappels sur la **complexité**.
- **Arbres** binaires : vocabulaire, propriétés, parcours.
- Arbres binaires de **recherche**,
- Arbres **équilibrés**.
- Autres sortes d'arbres,
- **Tas**.
- Algorithme de **compression** basé sur les arbres.
- ...

Algorithmes programmés en OCaml, parfois en C.

Plan de ce premier cours

 Voir le polycopié pour plus de détails.

- Arbres binaires : vocabulaire et propriétés
- Parcours récursif en profondeur

   Démo de comparaison en **C** et **Ocaml**.
- Complexité : définition, notation $O()$, rédaction de preuve.

L'arbre comme structure de données

- Apparaît naturellement : XML, arborescences de répertoires, ...
- Structure qui permet de ranger efficacement des données.
- Gain de temps spectaculaire par rapport à listes/piles/files.
  - **10000 ans** $\rightarrow$ quelques secondes.
- Cette UE est un prérequis pour algo des graphes.

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 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 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.

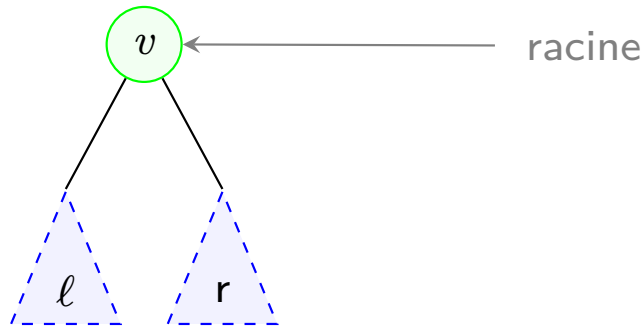
- arbre vide :

Arbres binaires : représentation graphique

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

- arbre vide :

- arbre non vide (v, ℓ, r) :



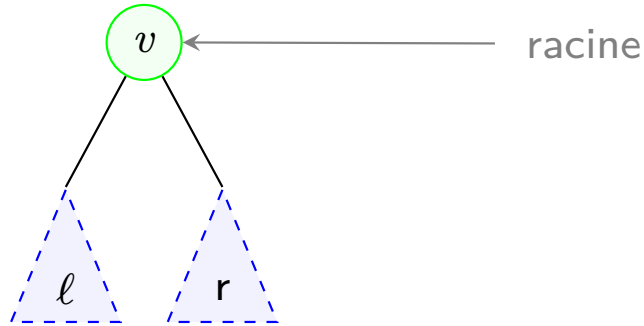
et le dessin continue, récursivement, pour ℓ et r.

Arbres binaires : représentation graphique

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

- arbre vide :

- arbre non vide (v, ℓ, r) :

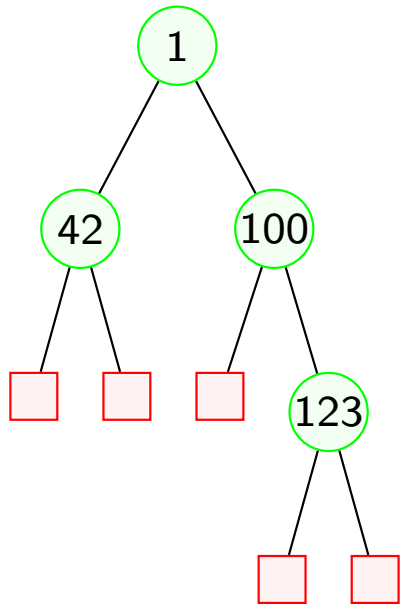


et le dessin continue, récursivement, pour ℓ et r.

Attention ! L'arbre vide n'est pas un nœud !

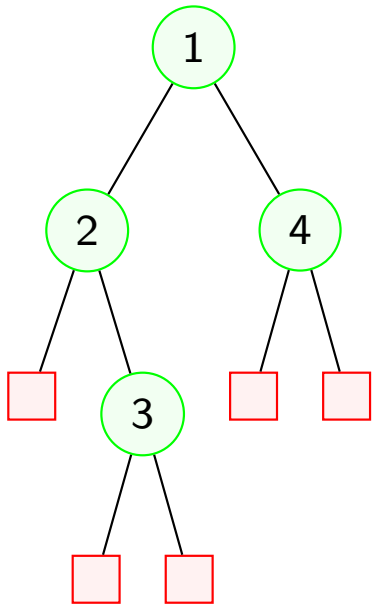
Exemple

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

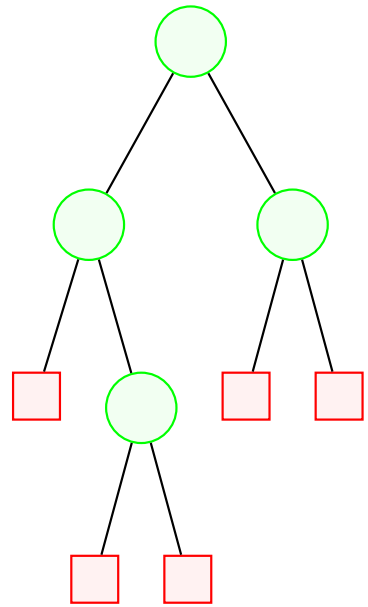


Squelette d'un arbre

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

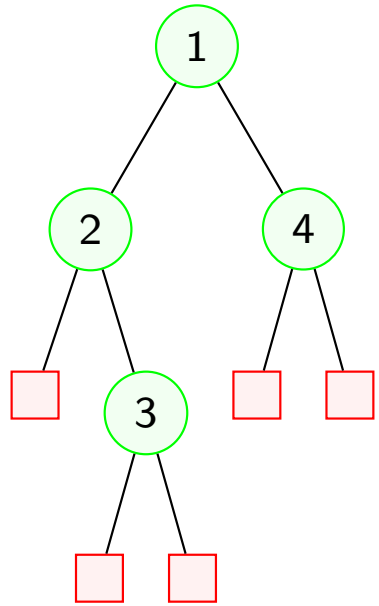


(a) Un arbre binaire

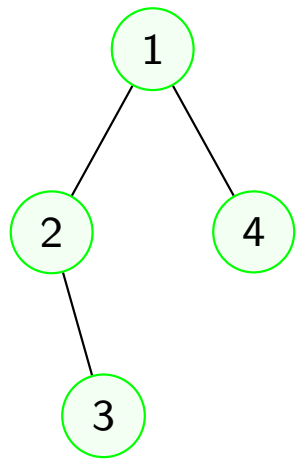


(b) Son squelette

Sous-arbres vides : inutiles !



(a) Un arbre binaire



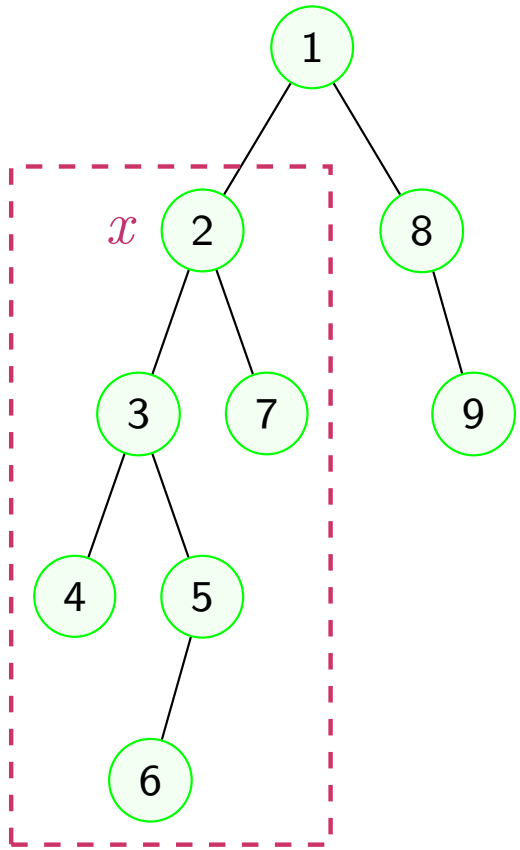
(b) Le même, sans dessiner les sous-arbres vides

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.

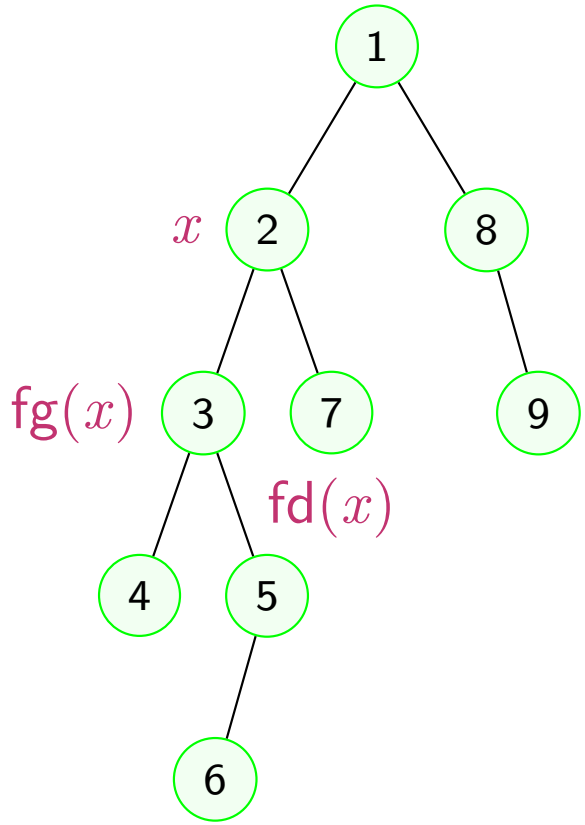
Sous-arbre enraciné en un nœud x

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



Fils gauche, fils droit d'un nœud x

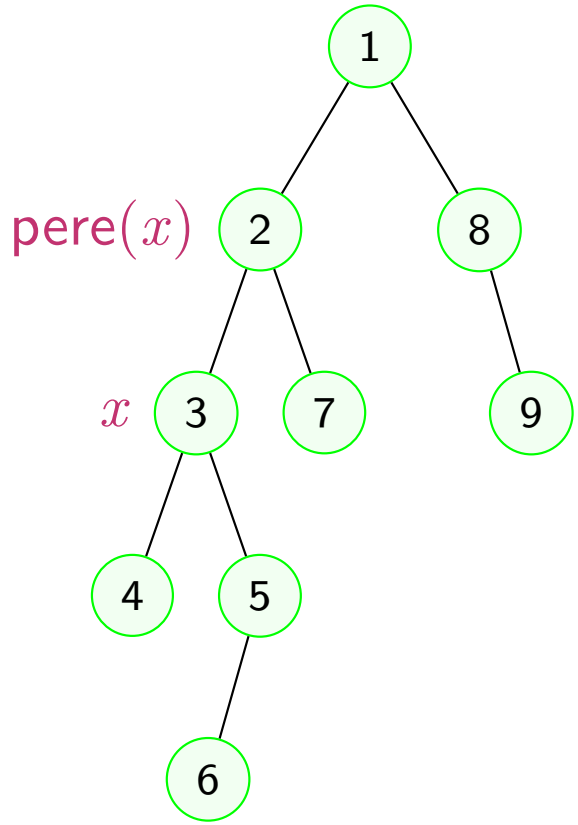
Fils gauche du nœud x, noté $\mathrm{fg}(x)$ = racine du sous-arbre gauche.



- 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

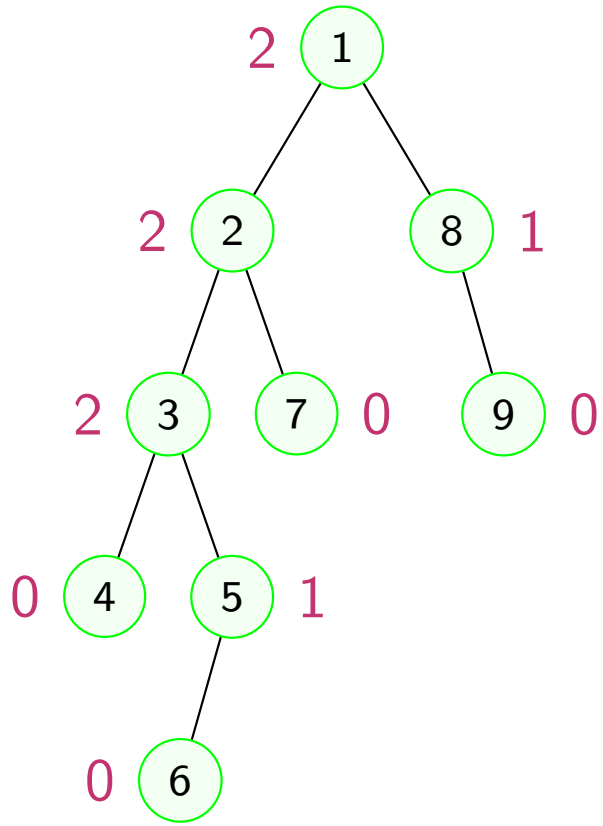
Le père d'un fils de x est x.



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

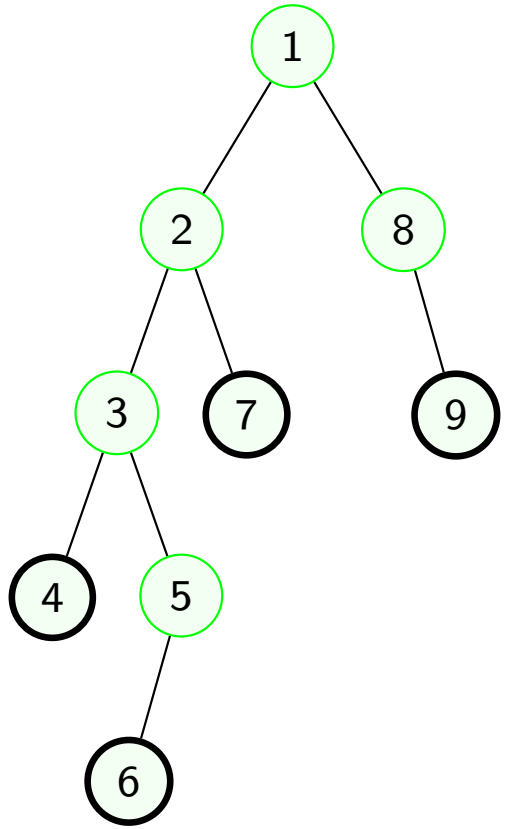
Arité d'un nœud

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



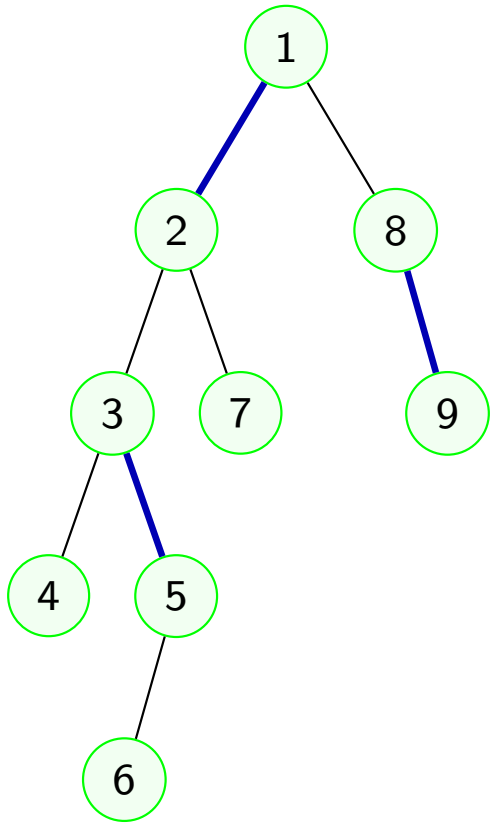
Feuille

Feuille = nœud dont l'arité est **0**.



Arête

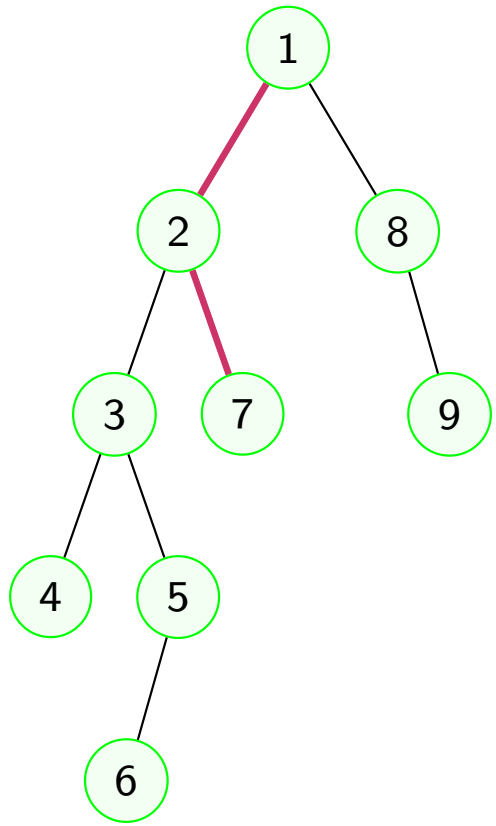
Relie un fils et son père.



Trois arêtes de cet arbre, en bleu.

Branche

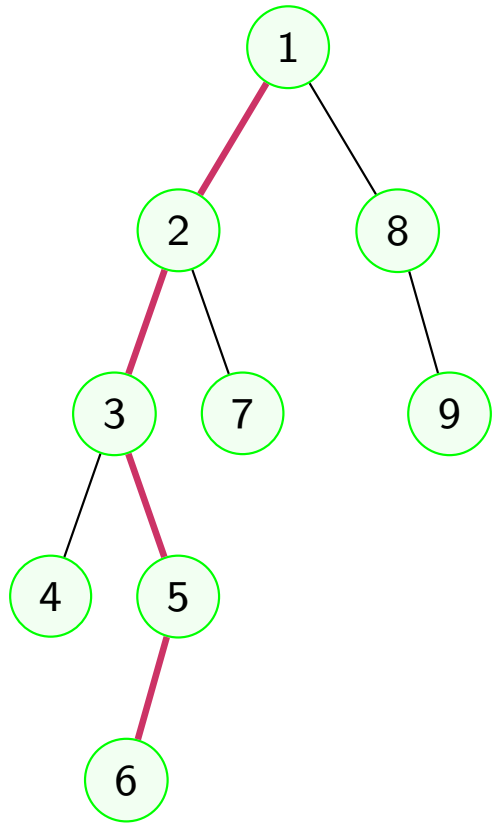
Chemin de la racine à une feuille.



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

Hauteur d'un arbre

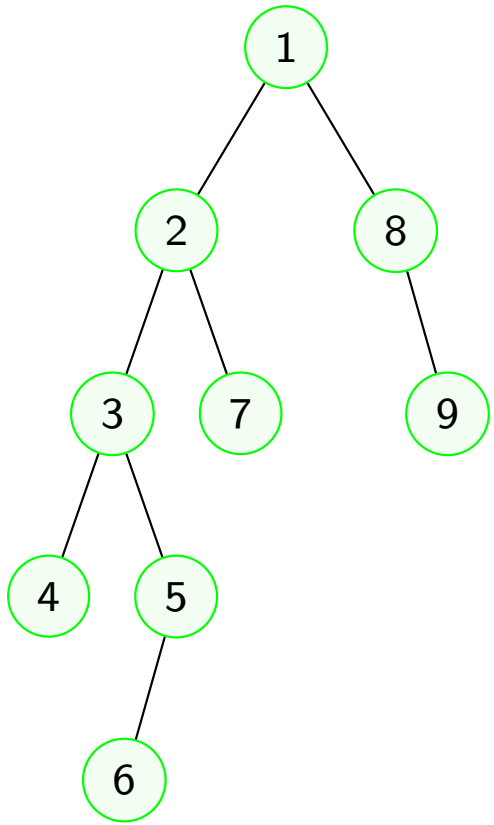
Longueur maximale d'une branche.



La hauteur de cet arbre est **4**.

Taille d'un arbre

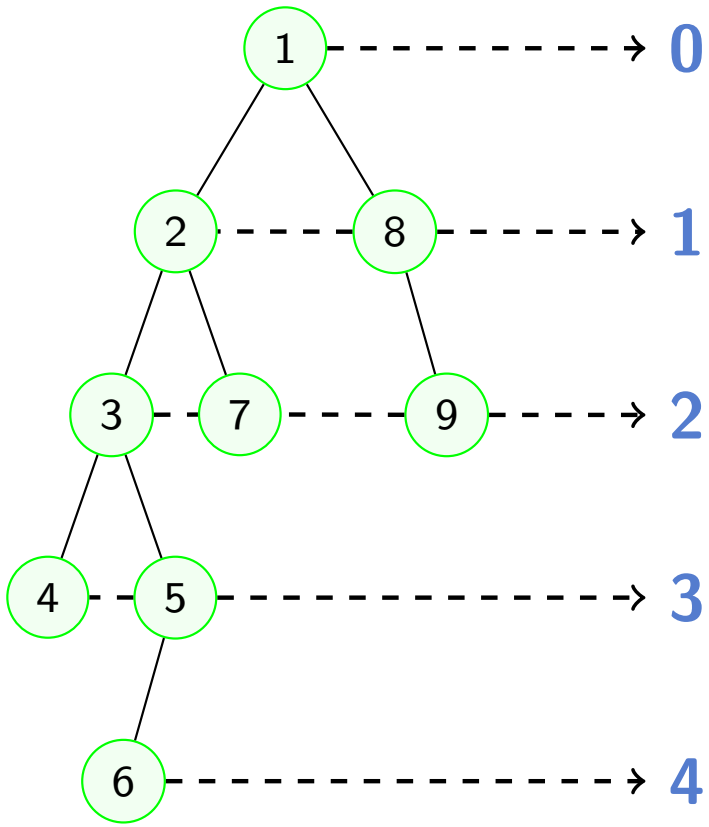
Nombre de nœuds.



Cet arbre a comme taille **9**.

Profondeur (ou niveau) d'un nœud

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



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

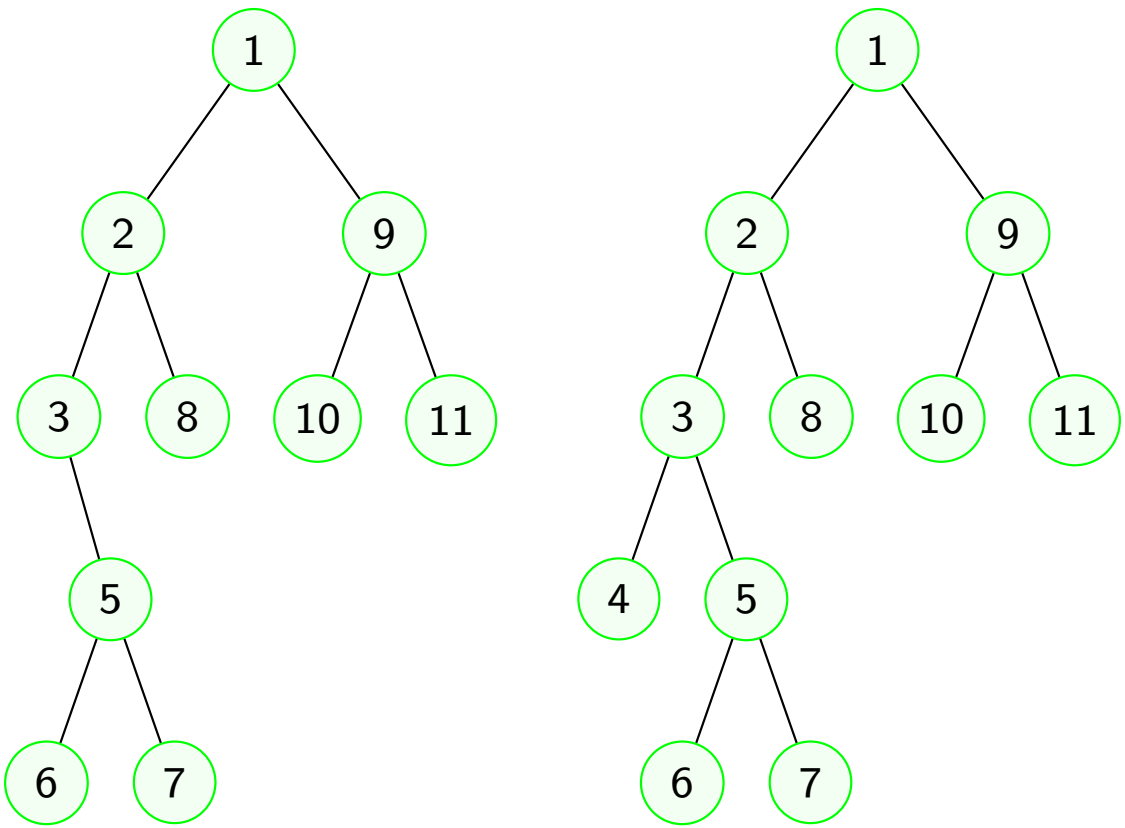
Arbres binaires particuliers

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 ».

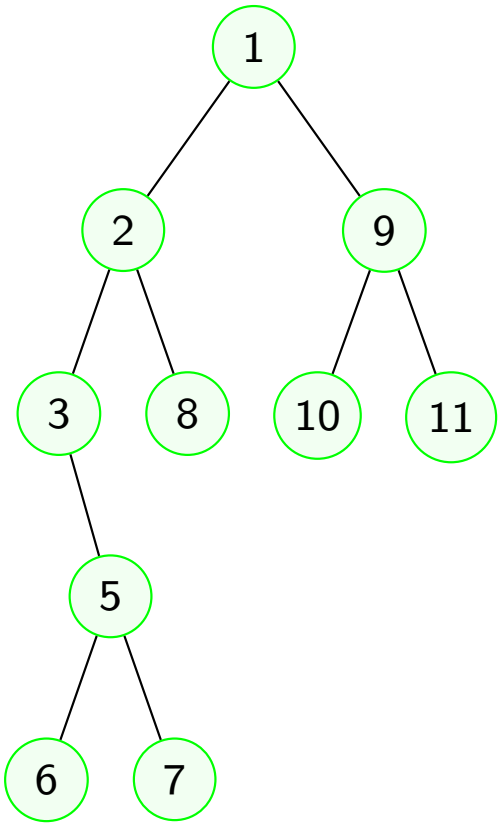
Remarque : tout arbre parfait est quasi-parfait.

Arbre complet

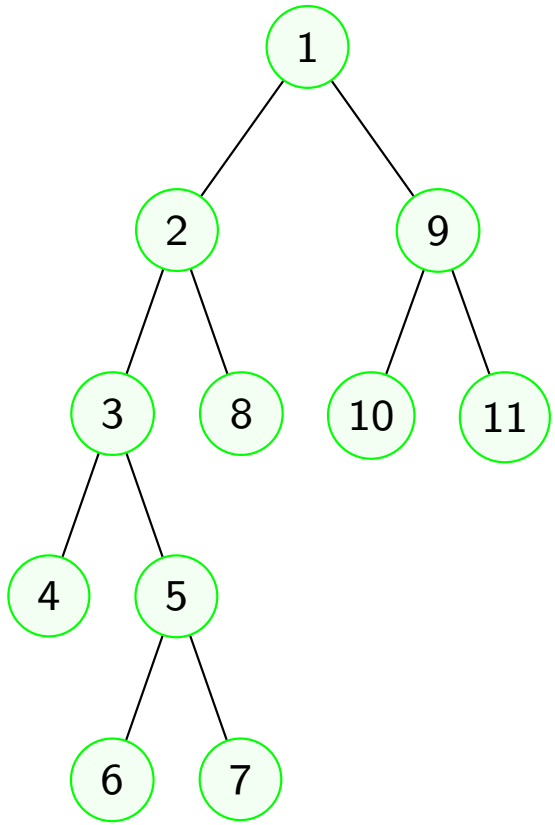


Arbre complet

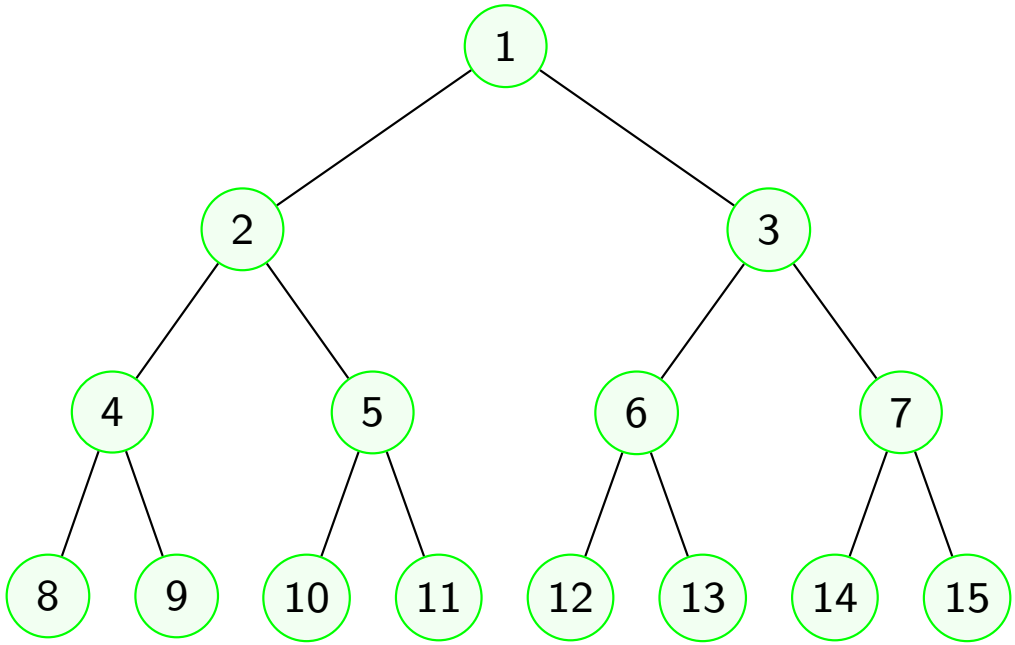
Non complet



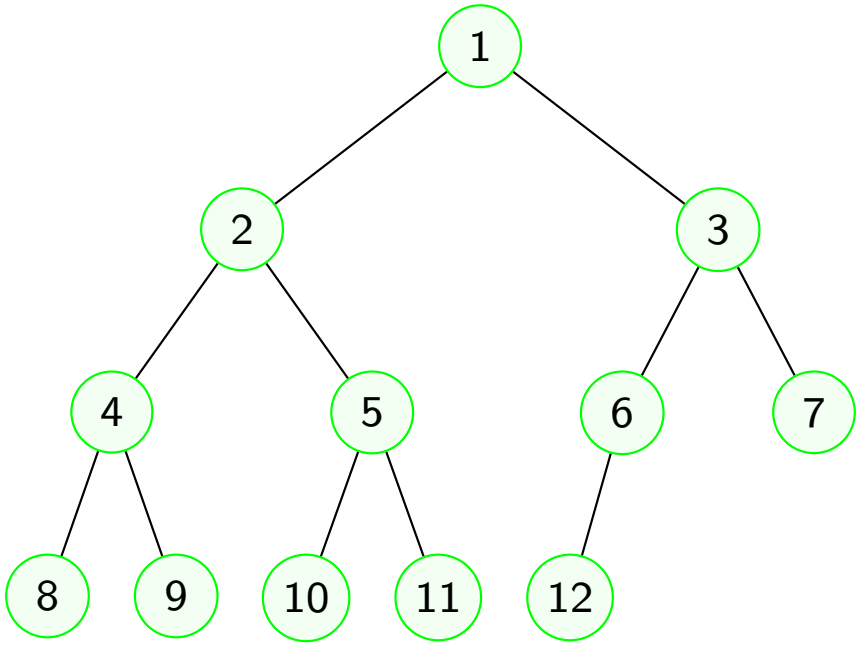
Complet



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.

Pourquoi une preuve formelle et comment rédiger ?

Tout arbre binaire non vide a au moins une feuille

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

- Comme t est non vide, on a $n \geqslant 1$.

Tout arbre binaire non vide a au moins une feuille

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

- Comme t est non vide, on a $n \geqslant 1$.
- Si $n = 1$, 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 $n(t) = n$ de l'arbre t.

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

Tout arbre binaire non vide a au moins une feuille

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

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

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

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

Tout arbre binaire non vide a au moins une feuille

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

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

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

    où $n(\ell) =$ taille de ℓ et $n(r) =$ 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 $n(t) = n$ de l'arbre t.

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

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

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

Tout arbre binaire non vide a au moins une feuille

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

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

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

    où $n(\ell) =$ taille de ℓ et $n(r) =$ taille de r.
  - Comme $n > 1$, soit ℓ soit r n'est pas vide, par exemple ℓ.
  - D'après l'équation (1), on a $n(\ell) \leqslant n - 1$.
  - 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 $n(t) = n$ de l'arbre t.

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

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

    où $n(\ell) =$ taille de ℓ et $n(r) =$ taille de r.
  - Comme $n > 1$, soit ℓ soit r n'est pas vide, par exemple ℓ.
  - D'après l'équation (1), on a $n(\ell) \leqslant n - 1$.
  - 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,
- $n(t)$ 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 $h(t) + 1 \leqslant n(t) \leqslant 2^{h(t)+1} - 1$.
2. L'arité de tous les nœuds internes de t est 1 si et seulement si

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

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

Propriétés (se montrent par récurrence)

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

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

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

$$n(t) = 2^{h(t)+1} - 1.$$

4. Si t est un arbre **complet**, on a $i(t) =$

Propriétés (se montrent par récurrence)

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

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

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

$$n(t) = 2^{h(t)+1} - 1.$$

4. Si t est un arbre **complet**, on a $i(t) = \ell(t) - 1$.

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

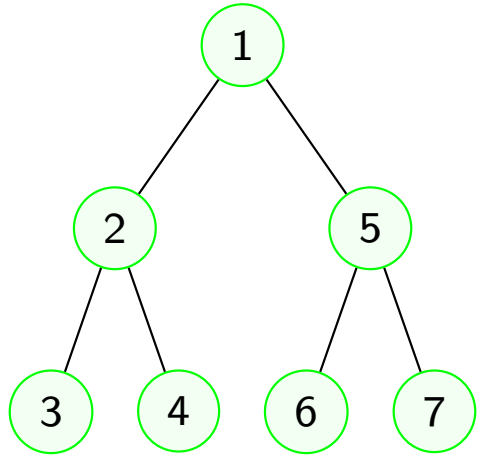


FIGURE – P2 : Arbre parfait de hauteur 2

Dans quel ordre les nœuds sont-ils traités ?

Parcours prefixe, infixe, postfixe : exemple

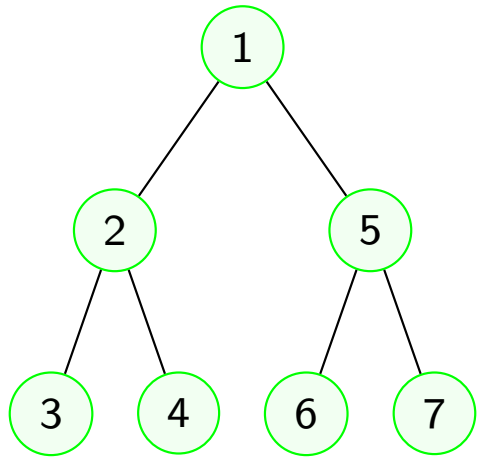


FIGURE – P2 : Arbre parfait de hauteur 2

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 !

Algorithmique des structures arborescentes
Algorithmique & programmation fonctionnelle

L2 Info et Math-info, 2019–20

Marc Zeitoun

30 janvier 2020

Moodle

- **Lire attentivement** les définitions (polycopié, diapos).
- Se référer aux énoncés **sous Moodle** uniquement.
- **Supprimer** la définition initiale (et incorrecte) des fonctions !

Moodle

- **Lire attentivement** les définitions (polycopié, diapos).
- Se référer aux énoncés **sous Moodle** uniquement.
- **Supprimer** la définition initiale (et incorrecte) des fonctions !
- **NE PAS UTILISER L'ÉDITEUR MOODLE SVP !**
  (sauf si vous aimez perdre votre temps).

Plan

Complexité en temps d'un algorithme

Il est souhaitable qu'un algorithme soit (correct et) **efficace**.
La **complexité en temps** d'un algorithme **mesure** son **efficacité**.

3 points à comprendre :

1. Évaluer l'efficacité sur **une** **entrée donnée**.

Complexité en temps d'un algorithme

Il est souhaitable qu'un algorithme soit (correct et) **efficace**.
La **complexité en temps** d'un algorithme **mesure** son **efficacité**.

3 points à comprendre :

1. Évaluer l'efficacité sur **une** **entrée donnée**.
2. Évaluer la pire complexité sur **toutes** **les entrées de taille** n.

Complexité en temps d'un algorithme

Il est souhaitable qu'un algorithme soit (correct et) **efficace**.
La **complexité en temps** d'un algorithme **mesure** son **efficacité**.

3 points à comprendre :

1. Évaluer l'efficacité sur **une** **entrée donnée**.
2. Évaluer la pire complexité sur **toutes** **les entrées de taille** n.
3. Gérer et **simplifier** les calculs : notation **O()**.

Efficacité d'un algorithme sur une entrée

But : Estimer la rapidité d'un algorithme **sur** **une entrée**.

Efficacité d'un algorithme sur une entrée

But : Estimer la rapidité d'un algorithme **sur** **une entrée**.

- L'estimation ne se fait pas en secondes (ou millisecondes).
- On n'utilise **pas d'unité**. **Pourquoi ?**

Efficacité d'un algorithme sur une entrée

But : Estimer la rapidité d'un algorithme **sur** **une entrée**.

- L'estimation ne se fait pas en secondes (ou millisecondes).
- On n'utilise **pas d'unité**. **Pourquoi ?**
- On estime le nombre d'instructions simples : opérations arithmétiques $+$, $*$, $-$, $/$, les affectations, les comparaisons, etc.

Efficacité d'un algorithme sur une entrée

But : Estimer la rapidité d'un algorithme **sur** **une entrée**.

- L'estimation ne se fait pas en secondes (ou millisecondes).
- On n'utilise **pas d'unité**. **Pourquoi ?**
- On estime le nombre d'instructions simples : opérations arithmétiques +, ∗, −, /, les affectations, les comparaisons, etc.

- **Avantages**
  1. **Pas d'hypothèse** à faire sur la rapidité du processeur.

Efficacité d'un algorithme sur une entrée

But : Estimer la rapidité d'un algorithme **sur** **une entrée**.

- L'estimation ne se fait pas en secondes (ou millisecondes).
- On n'utilise **pas d'unité**. **Pourquoi ?**
- On estime le nombre d'instructions simples : opérations arithmétiques +, ∗, −, /, les affectations, les comparaisons, etc.

- **Avantages**
  1. **Pas d'hypothèse** à faire sur la rapidité du processeur.
  2. **Simplification** des calculs.

Efficacité d'un algorithme sur une entrée

But : Estimer la rapidité d'un algorithme **sur** **une entrée**.

- L'estimation ne se fait pas en secondes (ou millisecondes).
- On n'utilise **pas d'unité**. **Pourquoi ?**
- On estime le nombre d'instructions simples : opérations arithmétiques +, ∗, −, /, les affectations, les comparaisons, etc.

- **Avantages**
  1. **Pas d'hypothèse** à faire sur la rapidité du processeur.
  2. **Simplification** des calculs.
  3. Permet quand même d'obtenir de **bons ordres de grandeur**.

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
      return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
     return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
     return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Nombre d'appels récursifs ?

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
     return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Nombre d'appels récursifs ?

$k + 1$ de $g(k)$ à $g(0)$

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
      return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Nombre d'appels récursifs ?

$k + 1$ de $g(k)$ à $g(0)$

```
def h(k):
  if k <= 0:
      return 1
  x = h(k//2)
  if k % 2 == 0:
      return x*x
  return 2*x*x
```

Nombre de multiplications pour $h(k)$?

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
      return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Nombre d'appels récursifs ?

$k + 1$ de $g(k)$ à $g(0)$

```
def h(k):
  if k <= 0:
      return 1
  x = h(k//2)
  if k % 2 == 0:
      return x*x
  return 2*x*x
```

Nombre de multiplications pour $h(k)$?

$\leqslant 2\log_2(k)$ quand on fait 1 ou 2 multiplications, on divise k par 2

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
      return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Nombre d'appels récursifs ?

$k + 1$ de $g(k)$ à $g(0)$

```
def h(k):
  if k <= 0:
      return 1
  x = h(k//2)
  if k % 2 == 0:
      return x*x
  return 2*x*x
```

Nombre de multiplications pour $h(k)$?

$\leqslant 2\log_2(k)$ quand on fait 1 ou 2 multiplications, on divise k par 2

Nombre d'appels récursifs ?

Exemple : calcul de 2^k

```
def f(k):
  if k <= 0:
    return 1
  return f(k-1)+f(k-1)
```

Nombre d'additions pour $f(k)$?

$2^k - 1$ on n'additionne que des "1"

Nombre d'appels récursifs ?

$2^{k+1} - 1$ arbre d'appels parfait, hauteur k

```
def g(k):
  if k <= 0:
      return 1
  return  2 * g(k-1)
```

Nombre de multiplications pour $g(k)$?

k on multiplie "1" k fois par 2

Nombre d'appels récursifs ?

$k + 1$ de $g(k)$ à $g(0)$

```
def h(k):
  if k <= 0:
      return 1
  x = h(k//2)
  if k % 2 == 0:
      return x*x
  return 2*x*x
```

Nombre de multiplications pour $h(k)$?

$\leqslant 2\log_2(k)$ quand on fait 1 ou 2 multiplications, on divise k par 2

Nombre d'appels récursifs ?

$\leqslant \log_2(k)$.

DEMO !

Note d'après cours

Pour tester comme en cours, lancez `ipython` dans un terminal :

```
$ ipython
Python 3.8.1 (default, Jan 10 2020, 23:28:47)
...

# Écrire le code de f, g, h
In [1]: def f(k):
   ...:
In [2]: def g(k):
   ...:
In [3]: def h(k):
   ...:
In [4]: %timeit -n1 -r1 h(100000000)
461 ms ± 0 ns per loop
```

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

f_{k+1} Par récurrence

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

```
# calcule (f_k, f_{k+1})
def g(k):
  if k <= 0:
      return 1,1
  a,b = g(k-1)
  return  b,a+b
```

Nombre d'additions pour $g(k)$?

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

```
# calcule (f_k, f_{k+1})
def g(k):
  if k <= 0:
      return 1,1
  a,b = g(k-1)
  return  b,a+b
```

Nombre d'additions pour $g(k)$?

$k - 1$.

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

```
# calcule (f_k, f_{k+1})
def g(k):
  if k <= 0:
      return 1,1
  a,b = g(k-1)
  return  b,a+b
```

Nombre d'additions pour $g(k)$?

$k - 1$.

Nombre d'appels récursifs ?

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

```
# calcule (f_k, f_{k+1})
def g(k):
  if k <= 0:
      return 1,1
  a,b = g(k-1)
  return  b,a+b
```

Nombre d'additions pour $g(k)$?

$k - 1$.

Nombre d'appels récursifs ?

k.

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

```
# calcule (f_k, f_{k+1})
def g(k):
  if k <= 0:
      return 1,1
  a,b = g(k-1)
  return  b,a+b
```

Nombre d'additions pour $g(k)$?

$k - 1$.

Nombre d'appels récursifs ?

k.

Comment faire $\alpha \log_2(k)$ opérations ?

```
def h(k):
```

?

Exemple : calcul du k-ème nombre de Fibonacci f_k

```
def f(k):
  if k <= 1:
    return 1
  return f(k-1)+f(k-2)
```

Nombre d'additions pour f_k ?

$f_k - 1$ Pourquoi ?

Nombre d'appels récursifs ?

$f_{k+1} \geqslant (\sqrt{2})^k$ Par récurrence

```
# calcule (f_k, f_{k+1})
def g(k):
  if k <= 0:
      return 1,1
  a,b = g(k-1)
  return  b,a+b
```

Nombre d'additions pour $g(k)$?

$k - 1$.

Nombre d'appels récursifs ?

k.

```
def h(k):
    ?
```

Comment faire $\alpha \log_2(k)$ opérations ?

En montrant (et utilisant) :

$$\begin{pmatrix} f_k \\ f_{k+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_{k-1} \\ f_k \end{pmatrix}$$

Complexité en temps d'un algorithme

- Un algorithme est en général conçu pour pouvoir traiter **une infinité d'entrées**.
- Il met souvent plus de temps sur une grande entrée.

Complexité en temps d'un algorithme

- Un algorithme est en général conçu pour pouvoir traiter **une infinité d'entrées**.
- Il met souvent plus de temps sur une grande entrée.
- **Complexité en temps dans le cas le pire** d'un algorithme :

Fonction qui à $n \in \mathbb{N}$ associe le nombre **maximal** d'opérations élémentaires faites par l'algorithme sur les entrées **de taille** n.

Complexité en temps d'un algorithme

- Un algorithme est en général conçu pour pouvoir traiter **une infinité d'entrées**.
- Il met souvent plus de temps sur une grande entrée.
- **Complexité en temps dans le cas le pire** d'un algorithme :

Fonction qui à $n \in \mathbb{N}$ associe le nombre **maximal** d'opérations élémentaires faites par l'algorithme sur les entrées **de taille** n.

Rappel. Opérations souvent considérées comme élémentaires : affectation, arithmétique $+, -, *, /$, comparaisons d'entiers, etc.

Évaluer la complexité d'un algorithme

Comment évaluer une complexité ?

Exemple : **tri par insertion**

Évaluer la complexité d'un algorithme

Comment évaluer une complexité ?

Exemple : **tri par insertion**

Sites de **visualisation** d'exécutions d'algorithmes.

- `https://visualgo.net/en/`.
  - `https://visualgo.net/en/sorting`
- `https://www.cs.usfca.edu/~galles/visualization`
  - `https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html`

Nombreux algorithmes sur les **arbres**.

Tri par insertion, version récursive

Suppose: j est une position légale du tableau t

```
1: Tri-Insertion(t, j):     // Trie t entre positions 0 et j
2: si 0 < j alors
3:   Tri-Insertion(t, j − 1)              // Appel récursif
4:   Insérer(t, j)                  // Insère t[j] à sa place
5: fin si
```

Tri par insertion, version récursive

Suppose: j est une position légale du tableau t

```
1: Tri-Insertion(t, j):     // Trie t entre positions 0 et j
2: si 0 < j alors
3:   Tri-Insertion(t, j − 1)              // Appel récursif
4:   Insérer(t, j)                  // Insère t[j] à sa place
5: fin si
```

Il faut d'abord évaluer la complexité de `Insérer(t, j)`.

Insertion d'un élément

Suppose: t déjà trié entre les positions 0 et j-1

```
Insérer(t, j):                          // Insère t[j] à sa place
clé = t[j]
tant que 0 < j et t[j - 1] > clé faire
  t[j]     = t[j - 1]
  t[j - 1] = clé
  j        = j - 1
fin tant que
```

Pour comprendre cet algorithme, le faire tourner pas à pas.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier `t`, dans le **cas le pire**, quand `j` $= n - 1$ (où $n =$ taille de `t`).

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier `t`, dans le **cas le pire**, quand `j` $= n-1$ (où $n =$ taille de `t`).
- Au pire, on ramène `t`$[n-1]$ en position 0 $\Rightarrow n-1$ échanges.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier `t`, dans le **cas le pire**, quand `j` $= n-1$ (où $n =$ taille de `t`).
- Au pire, on ramène `t`$[n-1]$ en position 0 $\Rightarrow n-1$ échanges.
- Relation vérifiée par c ?

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier `t`, dans le **cas le pire**, quand `j` $= n-1$ (où $n =$ taille de `t`).
- Au pire, on ramène `t`$[n-1]$ en position 0 $\Rightarrow n-1$ échanges.
- Relation vérifiée par c ?

$$c(n) = \underbrace{c(n-1)}_{\substack{\text{Appel récursif} \\ \text{(ligne 3 de Tri-Insertion)}}} + \underbrace{(n-1)}_{\substack{\text{nombre d'échanges} \\ \text{dans \textbf{Insérer}, au pire}}}$$

Remarque : la relation est obtenue en **suivant l'algorithme**.

Tri par insertion récursif : complexité

- $c(n) \stackrel{\text{def}}{=}$ nombre d'échanges pour trier `t`, dans le **cas le pire**, quand `j` $= n-1$ (où $n =$ taille de `t`).
- Au pire, on ramène `t`$[n-1]$ en position 0 $\Rightarrow n-1$ échanges.
- Relation vérifiée par c ?

$$c(n) = \underbrace{c(n-1)}_{\substack{\text{Appel récursif} \\ \text{(ligne 3 de Tri-Insertion)}}} + \underbrace{(n-1)}_{\substack{\text{nombre d'échanges} \\ \text{dans \textbf{Insérer}, au pire}}}$$

Remarque : la relation est obtenue en **suivant l'algorithme**.

- Par récurrence : $c(n) = \dfrac{n(n-1)}{2}$.

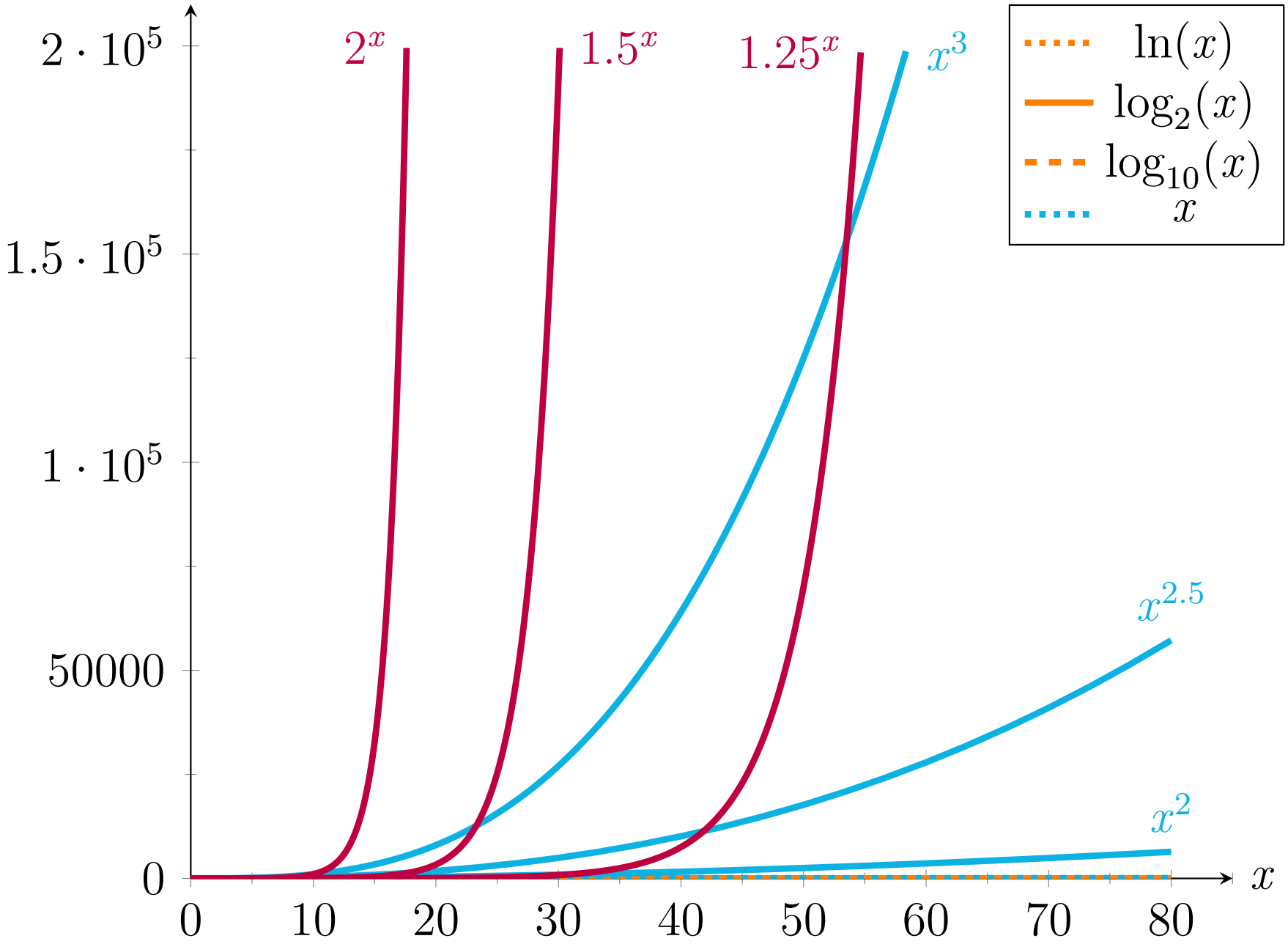
Gérer et simplifier les calculs : fonctions utiles

Dans ce cours, les fonctions de complexité sont construites à l'aide :

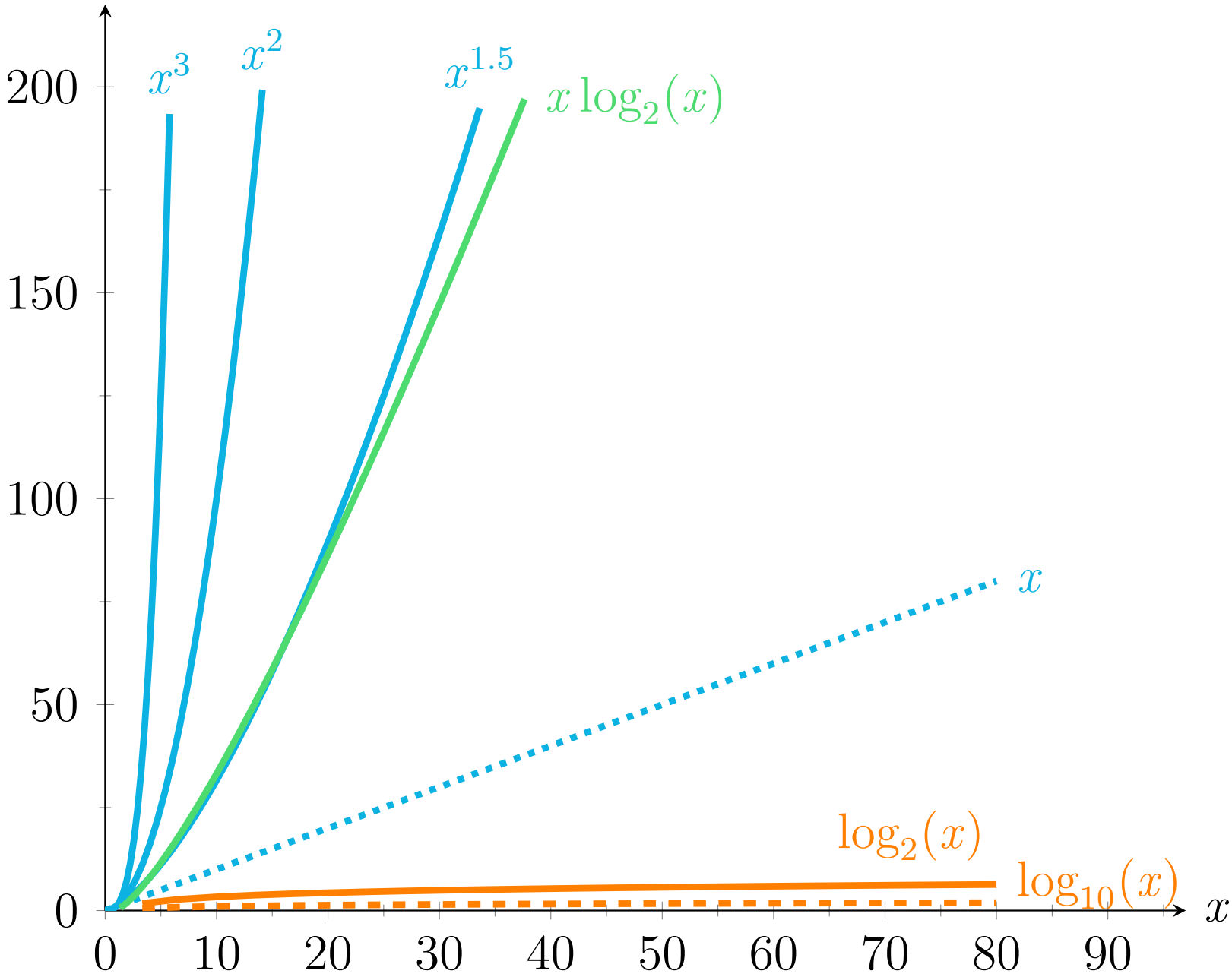
- des fonctions puissance $n \mapsto n^k$, souvent pour $k \in \mathbb{N}$,
- des fonctions logarithme $n \mapsto \log_2(n)$ ou $n \mapsto \log_{10}(n)$.
- des fonctions exponentielle $n \mapsto a^n$ pour $a > 1$, souvent entier.

Il faut avoir une intuition de leur rapidité de croissance.

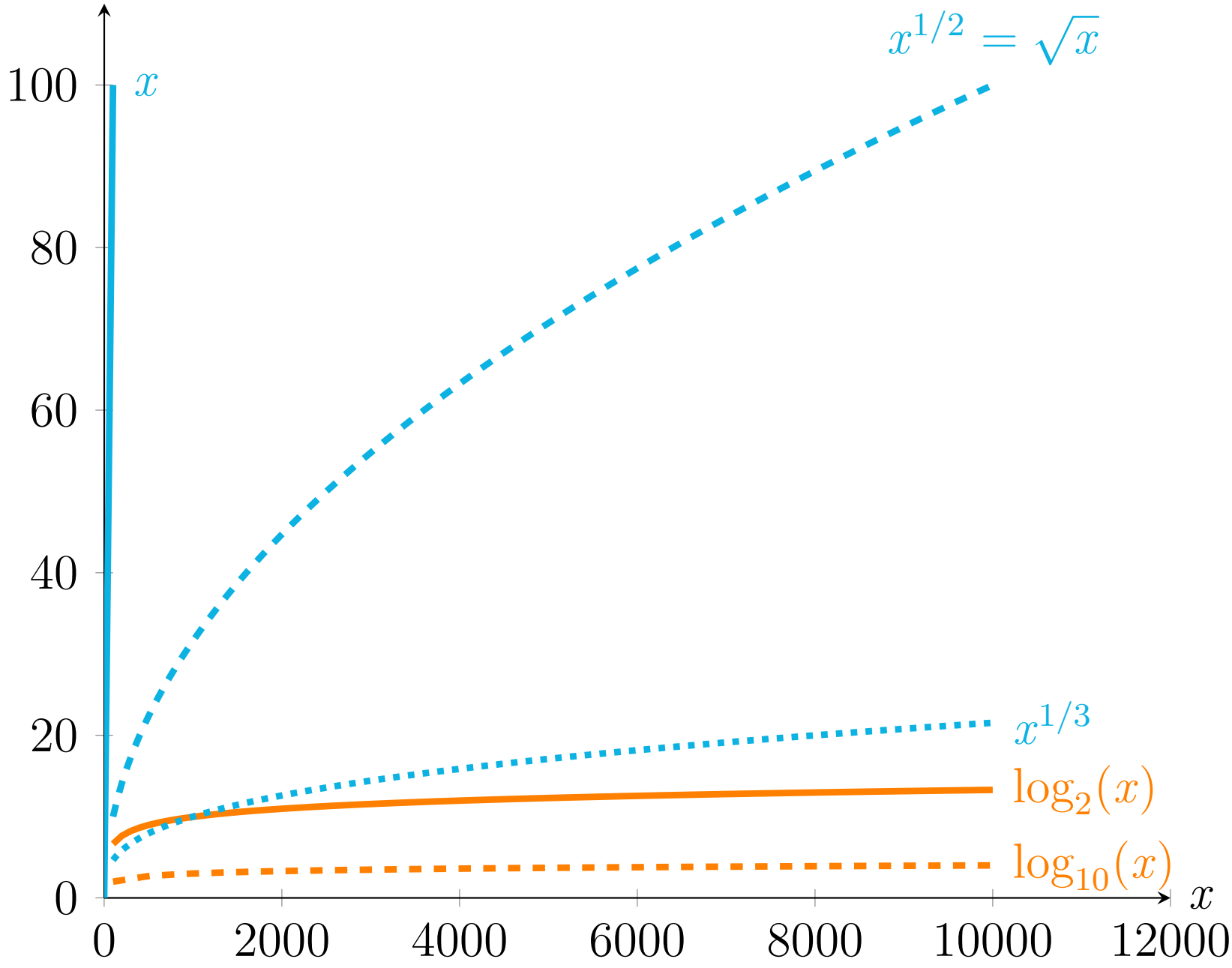
Croissance des fonctions classiques



Croissance des fonctions classiques (2)



Croissance des fonctions classiques (3)



Fonction logarithme

A retenir :

1. $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de x, moins 1).

   où $\lfloor x \rfloor$ = partie entière de x ($\lfloor 3,14 \rfloor = \lfloor 3,9 \rfloor = \lfloor 3 \rfloor = 3$).

   Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.

Fonction logarithme

A retenir :

1. $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de x, moins 1).

   où $\lfloor x \rfloor$ = partie entière de x ($\lfloor 3,14 \rfloor = \lfloor 3,9 \rfloor = \lfloor 3 \rfloor = 3$).

   Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.

Se prouve facilement à partir de la définition : $\log_{10}(10^x) = x$.

Fonction logarithme

A retenir :

1. $\lfloor \log_{10}(x) \rfloor$ = (nombre de chiffres de x, moins 1).
   où $\lfloor x \rfloor$ = partie entière de x ($\lfloor 3,14 \rfloor = \lfloor 3,9 \rfloor = \lfloor 3 \rfloor = 3$).
   Par exemple : $6 \leqslant \log_{10}(1234567) < 7$.

   Se prouve facilement à partir de la définition : $\log_{10}(10^x) = x$.

2. $\log_2(x) = \log_2(10) \cdot \log_{10}(x) \simeq 3,32 \log_{10}(x)$.
   - Les fonctions $\log_2$ et $\log_{10}$ sont proportionnelles.
     $\Rightarrow$ On parle souvent du « **log** » sans préciser la base.

3. $\log_2(x)$ croît **moins vite que toute fonction** x^a avec $a > 0$.
   - Formellement, $\lim\limits_{x \to \infty} \dfrac{\log_2^b(x)}{x^a} = 0$ si $a > 0$.

Fonction logarithme et exponentielle

À retenir :

1. La fonction **log** croit **très lentement**.

```
# x ~ âge estimé de l'univers, en secondes
In [1]: x = 14 * 10**9 * 365 * 24 * 3600
In [2]: from math import *
# x est énorme mais log2(x) est petit !
In [3]: log2(x)
Out[3]: 58.615204121749436
```

2. Inversement, la fonction 2^x croît **très rapidement**.

```
In [4]: 2**60
Out[4]: 1152921504606846976
# plus que l'âge estimé de l'univers en secondes
```

Notation O()

Si f, g sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, on écrit $f = O(g)$ si

$$\exists C > 0,\ \exists N > 0,\ \forall n, \quad n \geqslant N \implies f(n) \leqslant Cg(n)$$

“Quand n est grand, g domine f (à une constante multiplicative près)”

- **Exemples**
  - $2n + 1 = O(n)$ (**pourquoi ?**).

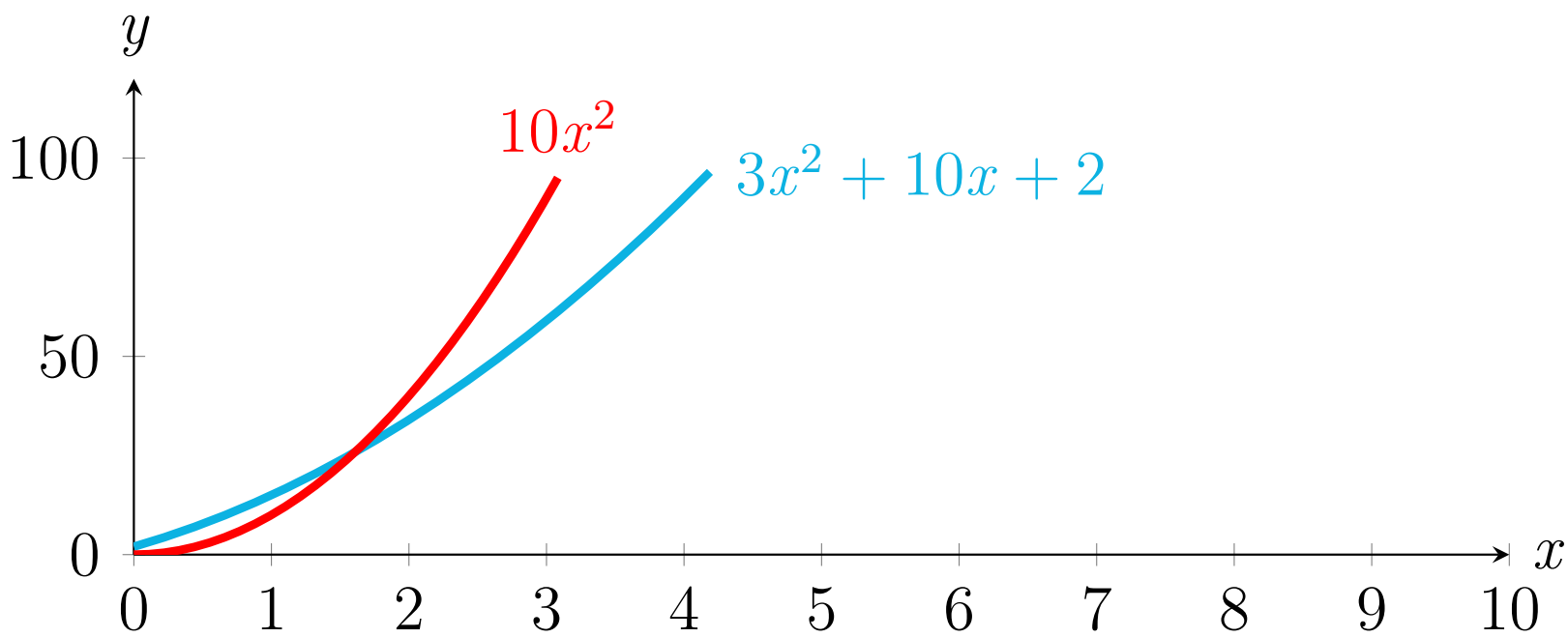
Notation O()

Si f, g sont des fonctions de $\mathbb{N}$ dans $\mathbb{N}$, on écrit $f = O(g)$ si

$$\exists C > 0,\ \exists N > 0,\ \forall n, \quad n \geqslant N \implies f(n) \leqslant Cg(n)$$

"Quand n est grand, g domine f (à une constante multiplicative près)"

- **Exemples**
  - $2n + 1 = O(n)$ (**pourquoi ?**).
  - $3n^2 + 10n + 2 = O(n^2)$ car $3n^2 + 10n + 2 \leqslant 10n^2$ si $n \geqslant 2$.



Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

$$7n^2 + 2n + 1 = O(n^2)$$

- De même, si $n \geqslant 1$:
  $25n^7 + 12n^3 + 4n\log_2(n) + 1 \leqslant 25n^7 + 12n^7 + 4n^7 + n^7 = 42n^7$,
  donc

Utilisation de la notation $O()$

Sert à **simplifier** des expressions de façon **raisonnable**.

- Si $n \geqslant 1$, on a $7n^2 + 2n + 1 \leqslant 7n^2 + 2n^2 + n^2 = 10n^2$ donc

$$7n^2 + 2n + 1 = O(n^2)$$

- De même, si $n \geqslant 1$:
  $25n^7 + 12n^3 + 4n\log_2(n) + 1 \leqslant 25n^7 + 12n^7 + 4n^7 + n^7 = 42n^7$,
  donc

$$25n^7 + 12n^3 + 4n\log_2(n) + 1 = O(n^7)$$

Utilisation de la notation $O()$ – (2)

- Si on a calculé une complexité en $n \log_2(n)$, on n'utiliserait pas $n \log_2(n) = O(n^2)$, même si c'est vrai, parce que
  - $n \log_2(n)$ est une expression simple,
  - on perd trop de précision en majorant par n^2.

Utilisation de la notation $O()$ – (2)

- Si on a calculé une complexité en $n \log_2(n)$, on n'utiliserait pas $n \log_2(n) = O(n^2)$, même si c'est vrai, parce que
  - $n \log_2(n)$ est une expression simple,
  - on perd trop de précision en majorant par n^2.

- Par contre, $\frac{1}{2}n^2 + 7n \log_2(n) + 12n + 8 \leqslant 28n^2$ si $n \geqslant 1$, donc

$$\frac{1}{2}n^2 + 7n \log_2(n) + 12n + 8 = O(n^2)$$

Le terme dominant étant $\frac{1}{2}n^2$, on ne perd pas l'ordre de grandeur en utilisant ici $n \log_2(n) = O(n^2)$.

Notation O() : formules utiles

- Complexité $O(1) =$ temps constant.
  Par exemple, multiplication de deux matrices 3×3
- Si $a < b$, on a $n^a = O(n^b)$.
- Si $a > 0$ et $c > 1$, on a $\log_c(n) = O(n^a)$.
  Un logarithme est dominé par une puissance.
- si $a > 0$ et $c > 1$, on a $n^a = O(c^n)$.
  Une puissance est dominée par une exponentielle.

Algorithmique des structures arborescentes
Algorithmique et programmation fonctionnelle

L2 Info et Math-info, 2019–20

Marc Zeitoun

6 février 2020

Plan

Plan

Arbres binaires de recherche

Min, Max, Recherche, Insertion, Suppression

Familles d'arbres équilibrés

Les arbres rouges et noirs

Les arbres rouges et noirs sont équilibrés

Les arbres binaires de recherche (ABR)

- Arbre dans lesquels les étiquettes sont **triées**.

Les arbres binaires de recherche (ABR)

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

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

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

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

Cette propriété ne se vérifie pas uniquement « localement ».

Les arbres binaires de recherche (ABR)

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

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

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

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

 Cette propriété ne se vérifie pas uniquement « localement ».

Un tel arbre est appelé **arbre binaire de recherche (ABR)**,

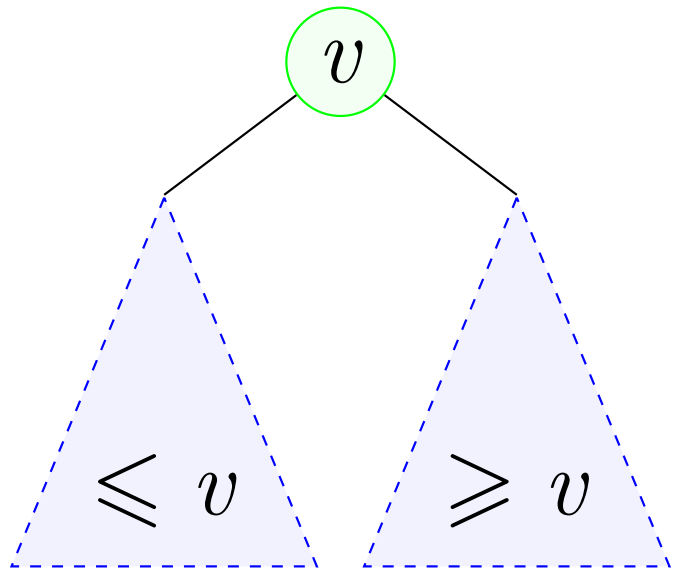
En anglais : **binary search tree (BST)**.

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.

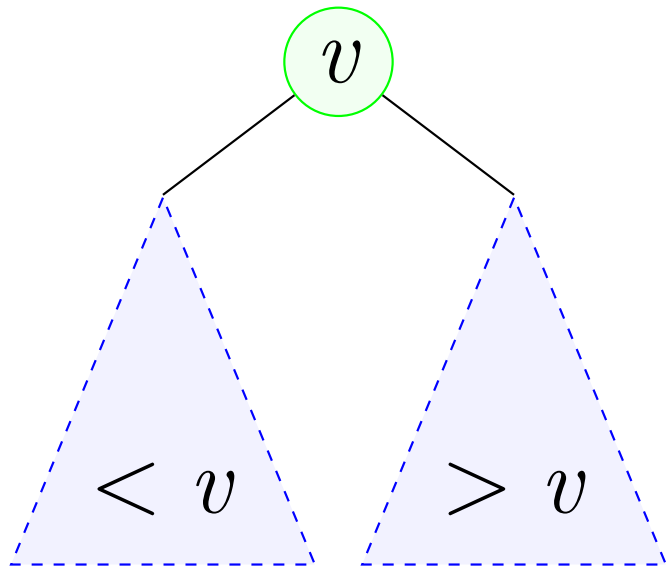


ABR

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

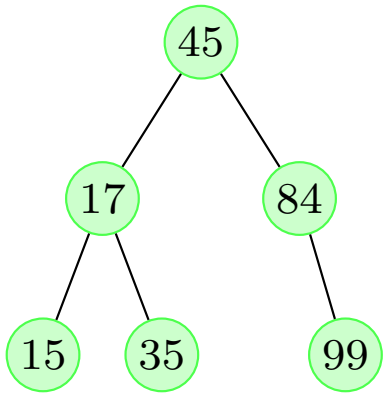
Pour **chaque nœud** v, elle implique :

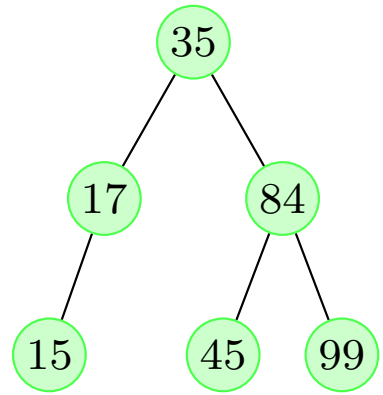
- **tous les nœuds** du sous-arbre gauche de v,
- **tous les nœuds** du sous-arbre droit de v.

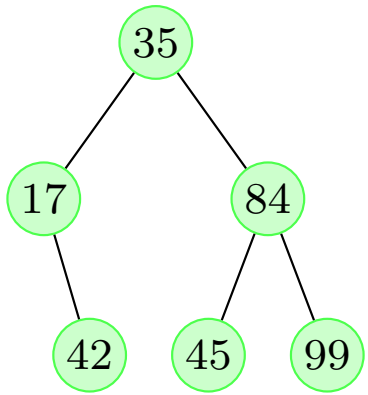


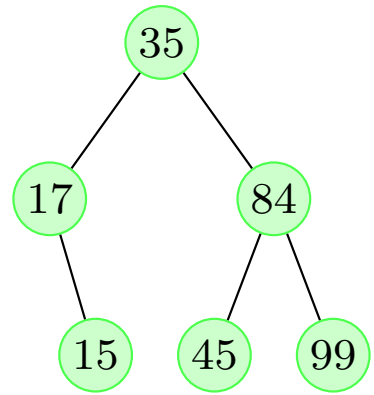
Dans ce cours, 2 nœuds auront toujours des clés différentes

ABR : exemples et non-exemples

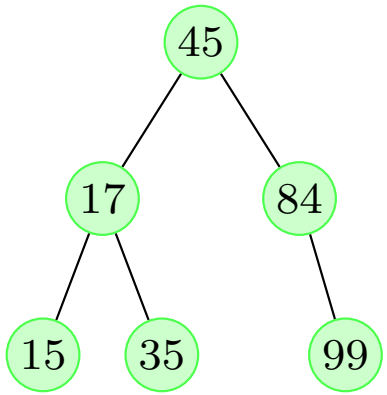




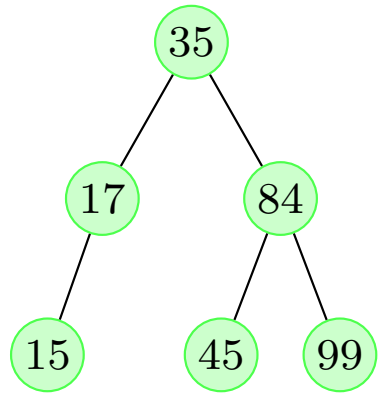


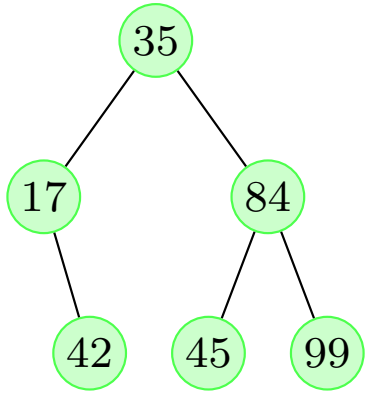


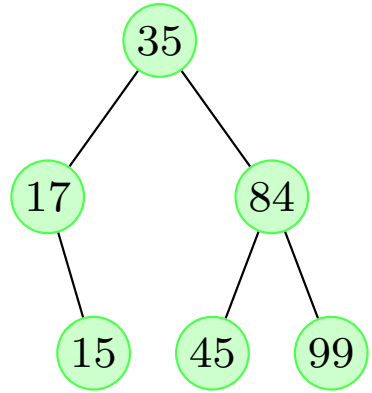
ABR : exemples et non-exemples



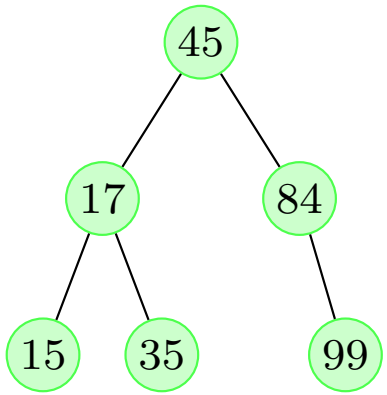
OUI



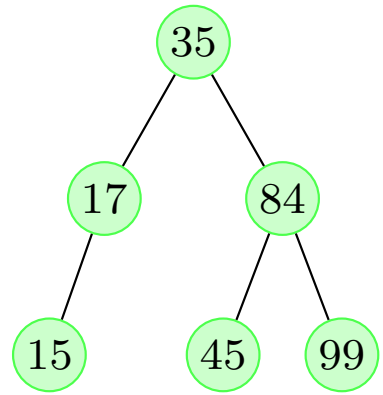




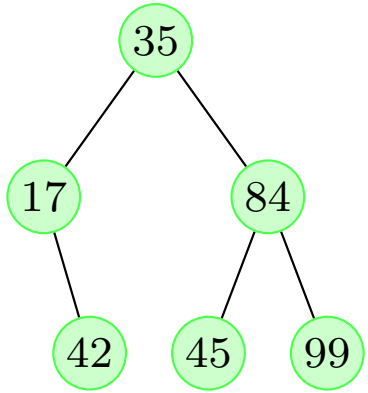
ABR : exemples et non-exemples

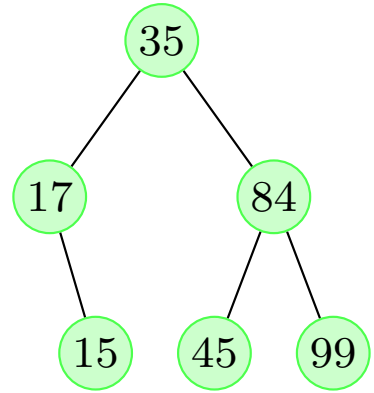


OUI

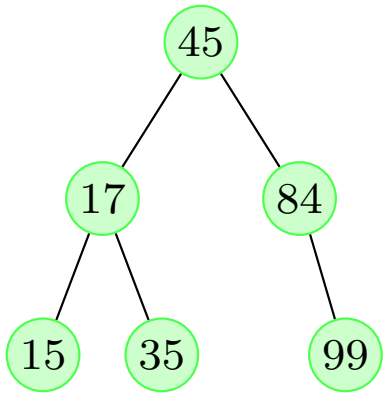


OUI

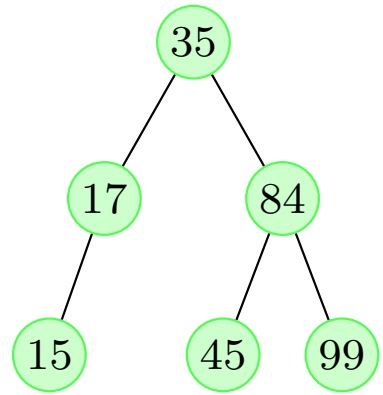




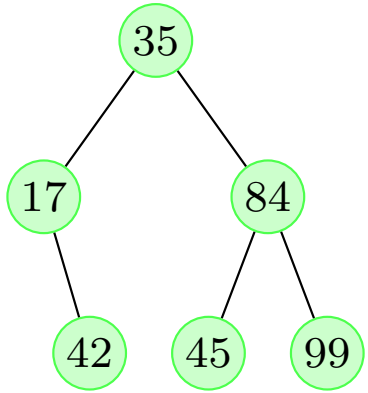
ABR : exemples et non-exemples



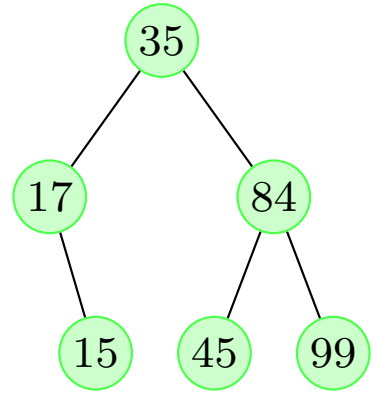
OUI



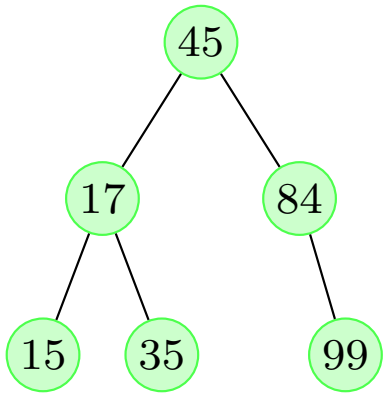
OUI



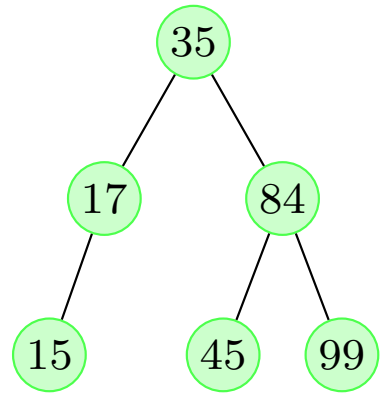
NON



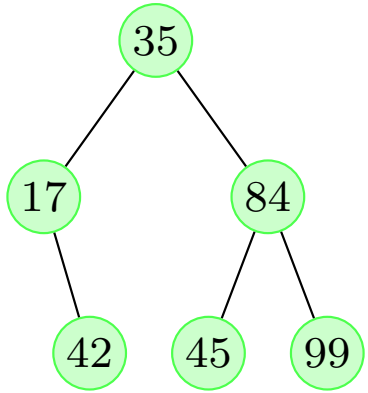
ABR : exemples et non-exemples



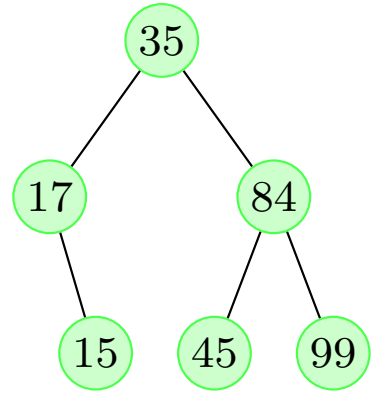
OUI



OUI



NON



NON

Représentation des multi-ensembles par ABR

- Multi-ensemble : fonction f d'un ensemble E dans $\mathbb{N}$.

  Intuition $f(e) = 4$ signifie : e est *présent en* 4 *exemplaires*.
- « Ensembles » pouvant contenir plusieurs fois la même valeur.
- Notation : $\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$.

Représentation des multi-ensembles par ABR

- Multi-ensemble : fonction f d'un ensemble E dans $\mathbb{N}$.

  Intuition $f(e) = 4$ signifie : e est *présent en* 4 *exemplaires*.
- « Ensembles » pouvant contenir plusieurs fois la même valeur.
- Notation : $\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$.
- L'ordre n'est pas important :

$$\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$$
$$=$$
$$\{\!\{5, 5, 1, 10, 5, 4, 7, 42, 1\}\!\}$$

Représentation des multi-ensembles par ABR

- Multi-ensemble : fonction f d'un ensemble E dans $\mathbb{N}$.

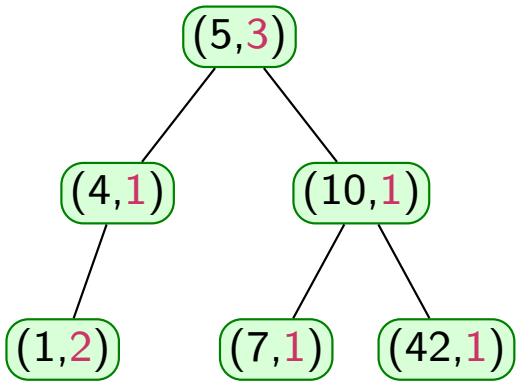
  Intuition $f(e) = 4$ signifie : e est *présent en* 4 *exemplaires*.
- « Ensembles » pouvant contenir plusieurs fois la même valeur.
- Notation : $\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\}$.
- L'ordre n'est pas important :

$$\{\!\{1, 1, 4, 5, 5, 5, 7, 10, 42\}\!\} = \{\!\{5, 5, 1, 10, 5, 4, 7, 42, 1\}\!\}$$

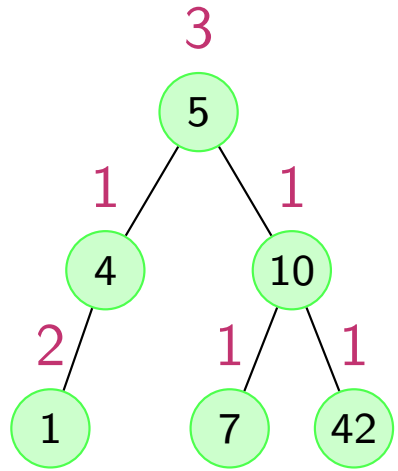
- On utilise les ABR pour représenter les multi-ensembles.
- Élément e en 4 exemplaires $\rightarrow$ nœud d'étiquette $(e, 4)$.

  $\Rightarrow$ Aucune clé n'apparaît dans plusieurs nœuds.

Notations



Couples avec clés et multiplicité



Multiplicité au dessus des nœuds

OCaml : `Node((5,3), Node(...), Node(...))`

Objectif

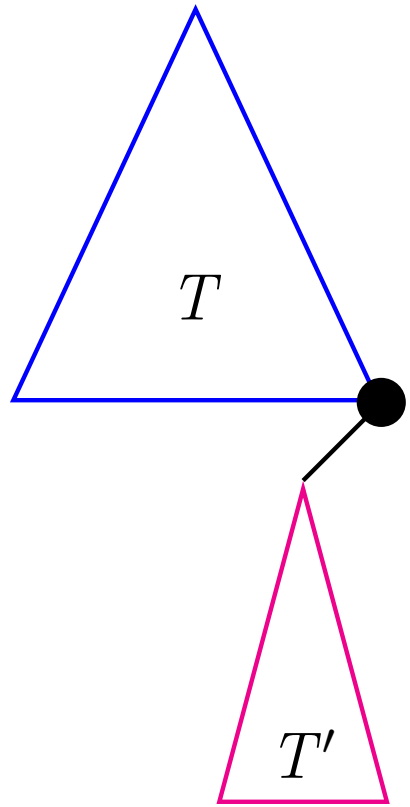
Manipuler des arbres représentant de **grands** multi-ensembles qui évoluent.

On veut pouvoir, rapidement :

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

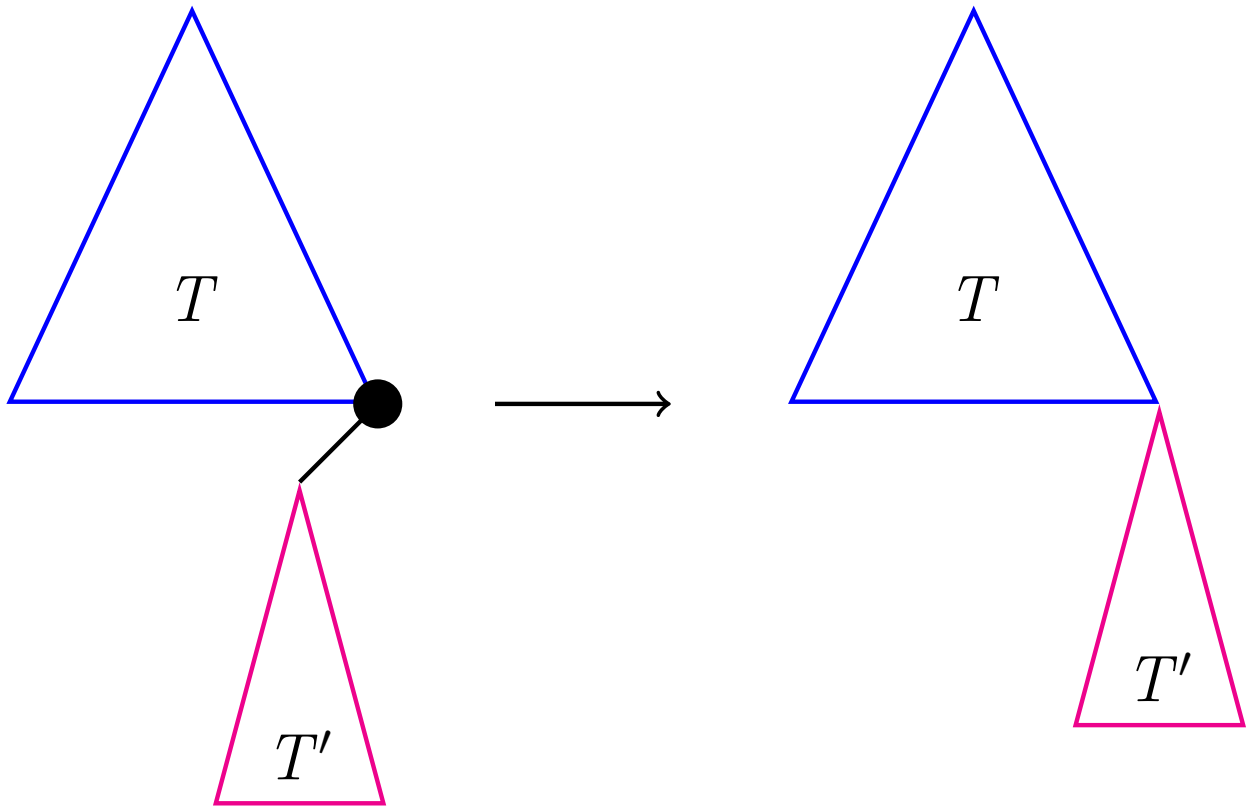
Élément maximum dans un ABR non vide

- 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

- En $O(1)$ une fois le nœud portant la clé maximale localisé.



Recherche d'un élément x

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

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

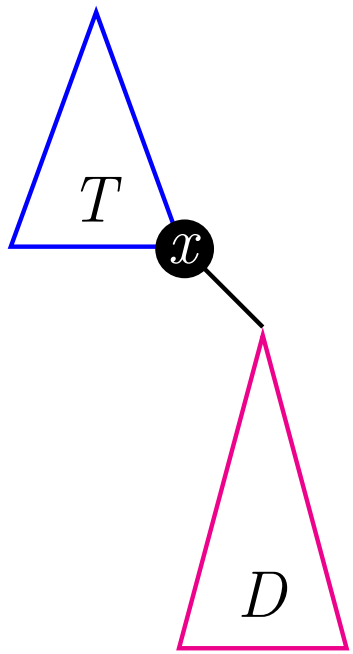
Insertion d'un élément

- Deux algorithmes classiques : par création d'une nouvelle feuille ou par ajout d'une nouvelle racine.
- **Dans ce cours** : par création d'une nouvelle feuille.
- Principe : naviguer jusqu'à
  - soit trouver un nœud portant la clé à insérer (et incrémenter sa multiplicité),
  - soit arriver jusqu'à un sous-arbre vide, et créer une feuille à cet emplacement.

Suppression d'un élément x de multiplicité 1

Principe :

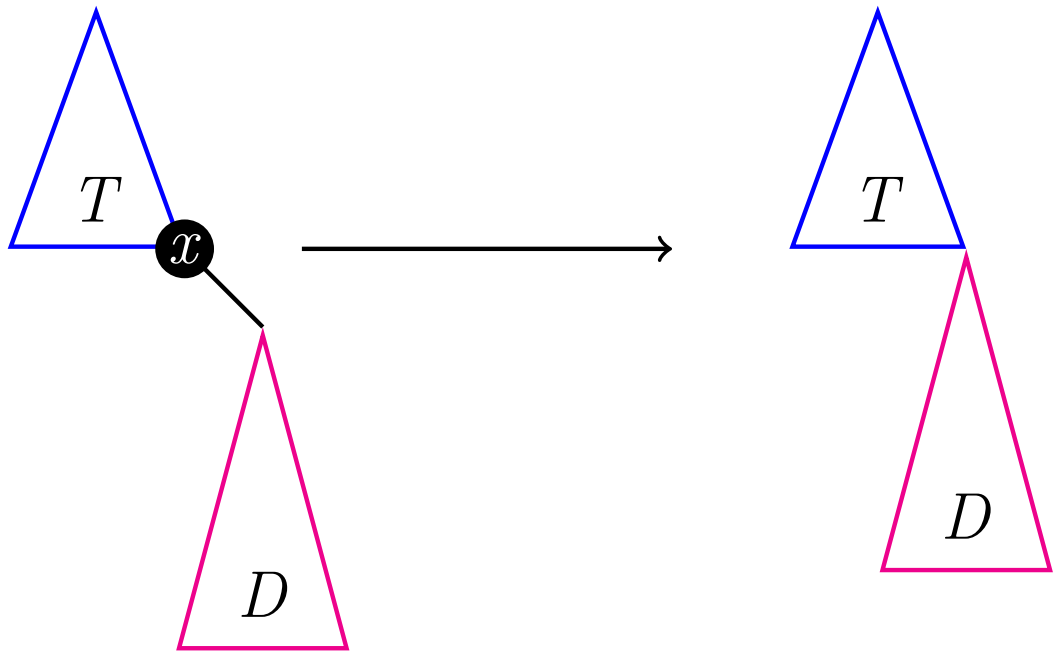
- Rechercher le nœud portant la clé x à supprimer.
- 1er cas : son sous-arbre gauche est vide :



Suppression d'un élément x de multiplicité 1

Principe :

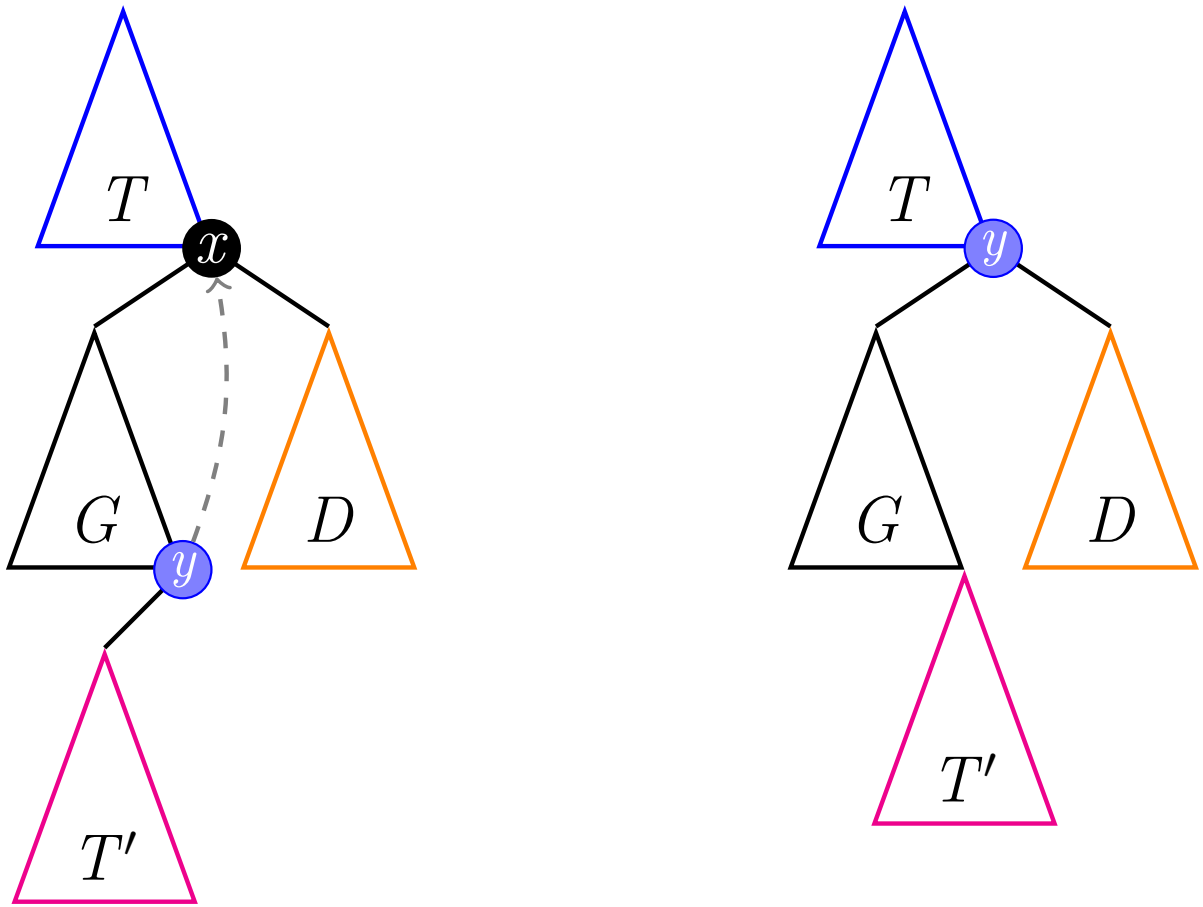
- Rechercher le nœud portant la clé x à supprimer.
- 1er cas : son sous-arbre gauche est vide :



Suppression d'un élément x de multiplicité 1

Principe :

- 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.



Plan

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR : $O(h)$ où h est la **hauteur** de l'arbre.

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

$$O(h) \text{ où } h \text{ est la } \textbf{hauteur} \text{ de l'arbre.}$$

Si n est le nombre de nœuds,

- Au pire, $h =$

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

$$O(h) \text{ où } h \text{ est la } \textbf{hauteur} \text{ de l'arbre.}$$

Si n est le nombre de nœuds,

- Au pire, $h = n - 1$.
- Au mieux, $h =$

ABR : Complexité recherche/insertion/suppression

- Complexité de recherche/insertion/suppression dans un ABR :

$$O(h) \text{ où } h \text{ est la } \textbf{hauteur} \text{ de l'arbre.}$$

Si n est le nombre de nœuds,

- Au pire, $h = n - 1$.
- Au mieux, $h = \log_2(n+1) - 1$.

Familles d'arbres équilibrés

- Maintenir dans les ABR

$$h = \alpha \log(n)$$

garantirait une complexité $O(\log(n))$ **pour ces opérations**.

Familles d'arbres équilibrés

- Maintenir dans les ABR

$$h = \alpha \log(n)$$

garantirait une complexité $O(\log(n))$ **pour ces opérations**.

Nouvel objectif

Maintenir $h = \alpha \log(n)$ pour un nombre $\alpha > 0$.

Plan

Les arbres rouges et noirs

Problème : si on insère 1, puis 2, 3, 4, ..., n, on obtient un arbre filiforme.

Idée :

- ajouter des conditions pour garantir $h = \alpha \log(n)$.
- maintenir ces conditions lors des insertions et suppressions.

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet**

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,

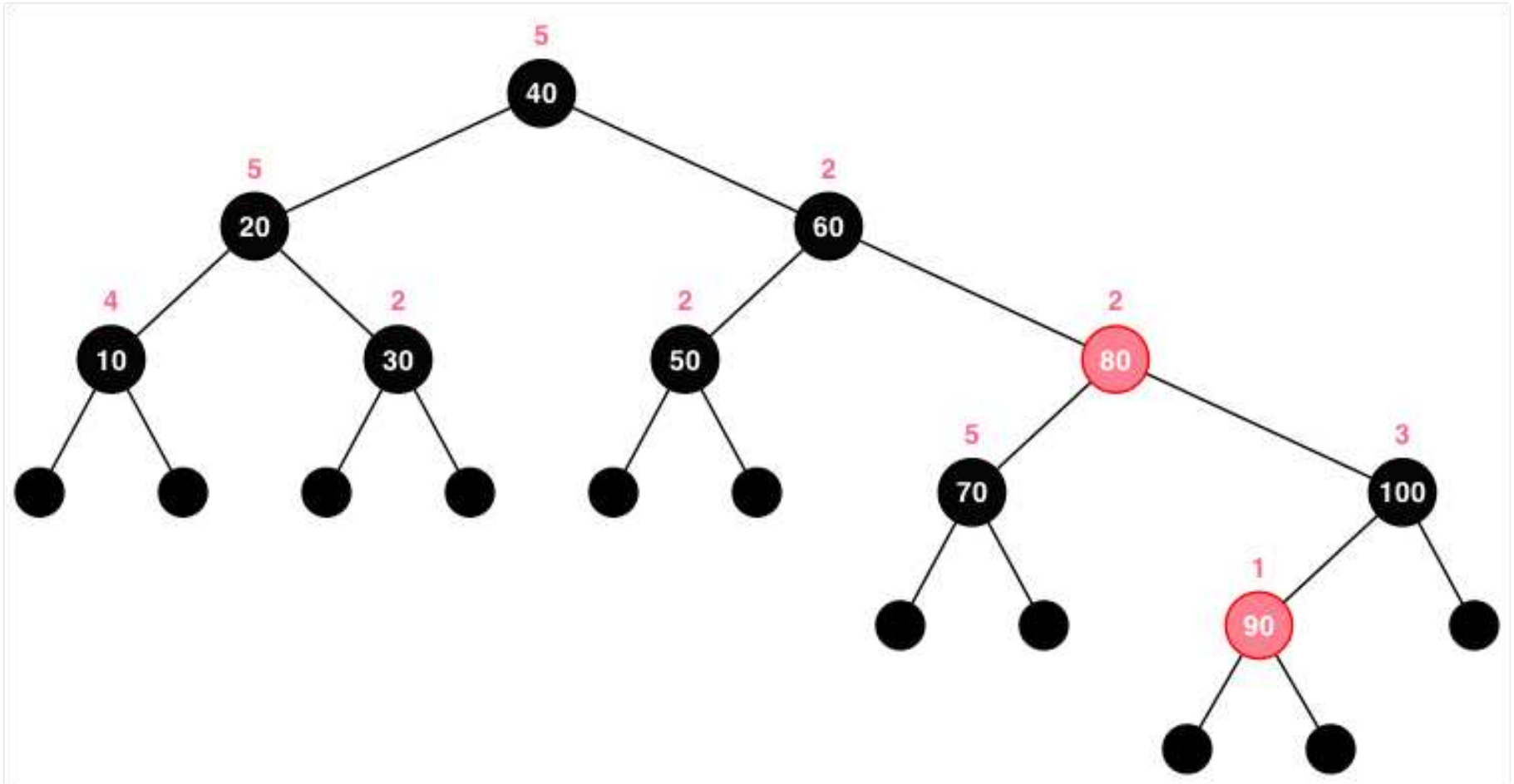
Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

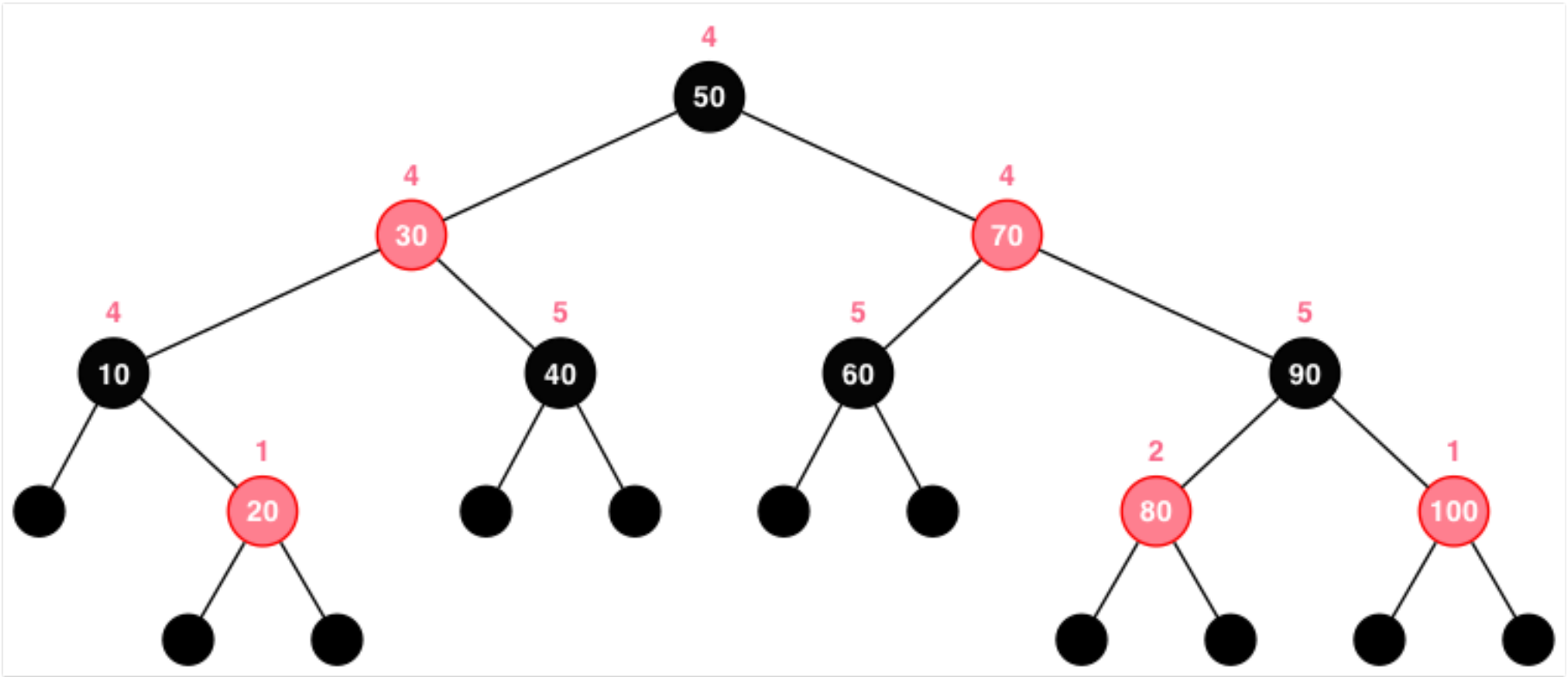
1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,
6. le père d'un nœud rouge est noir.
7. le nombre de nœuds noirs sur chaque branche est **constant**.

Hauteur noire = nombre de nœuds noirs sur chaque branche.

Arbres rouges et noirs : exemple 1



Arbres rouges et noirs : exemple 2



Plan

Les arbres rouges et noirs sont équilibrés

Preuve : sans récurrence. Soit t un arbre rouge et noir.

- $H =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre est complet et les branches ont au moins H nœuds.

  Donc tous les niveaux sont complets jusqu'à profondeur $H - 1$

Les arbres rouges et noirs sont équilibrés

Preuve : sans récurrence. Soit t un arbre rouge et noir.

- $H =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre est complet et les branches ont au moins H nœuds.

  Donc tous les niveaux sont complets jusqu'à profondeur $H - 1$
- Donc il y a au minimum $2^H - 1$ nœuds :

  $$2^H - 1 \leqslant n, \text{ donc : } \boldsymbol{H \leqslant \log_2(n+1).}$$

Les arbres rouges et noirs sont équilibrés

Preuve : sans récurrence. Soit t un arbre rouge et noir.

- $H =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre est complet et les branches ont au moins H nœuds.

  Donc tous les niveaux sont complets jusqu'à profondeur $H - 1$
- Donc il y a au minimum $2^H - 1$ nœuds :

  $$\mathbf{2^H - 1 \leqslant n}, \text{ donc : } \mathbf{H \leqslant \log_2(n+1).}$$
- Une branche a H nœuds noirs et au maximum $H - 1$ nœuds rouges : $\bullet - \bullet - \bullet - \bullet - \cdots - \bullet - \bullet - \bullet$.
- Donc $h \leqslant 2(H-1)$ et en utilisant l'inégalité ci-dessus :

  $$h \leqslant 2(\log_2(n+1) - 1).$$

Algorithmique des structures arborescentes
Algorithmique et programmation fonctionnelle

L2 Info et Math-info, 2019–20

Marc Zeitoun

20 février 2020

Plan

Plan

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet**

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,

Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,

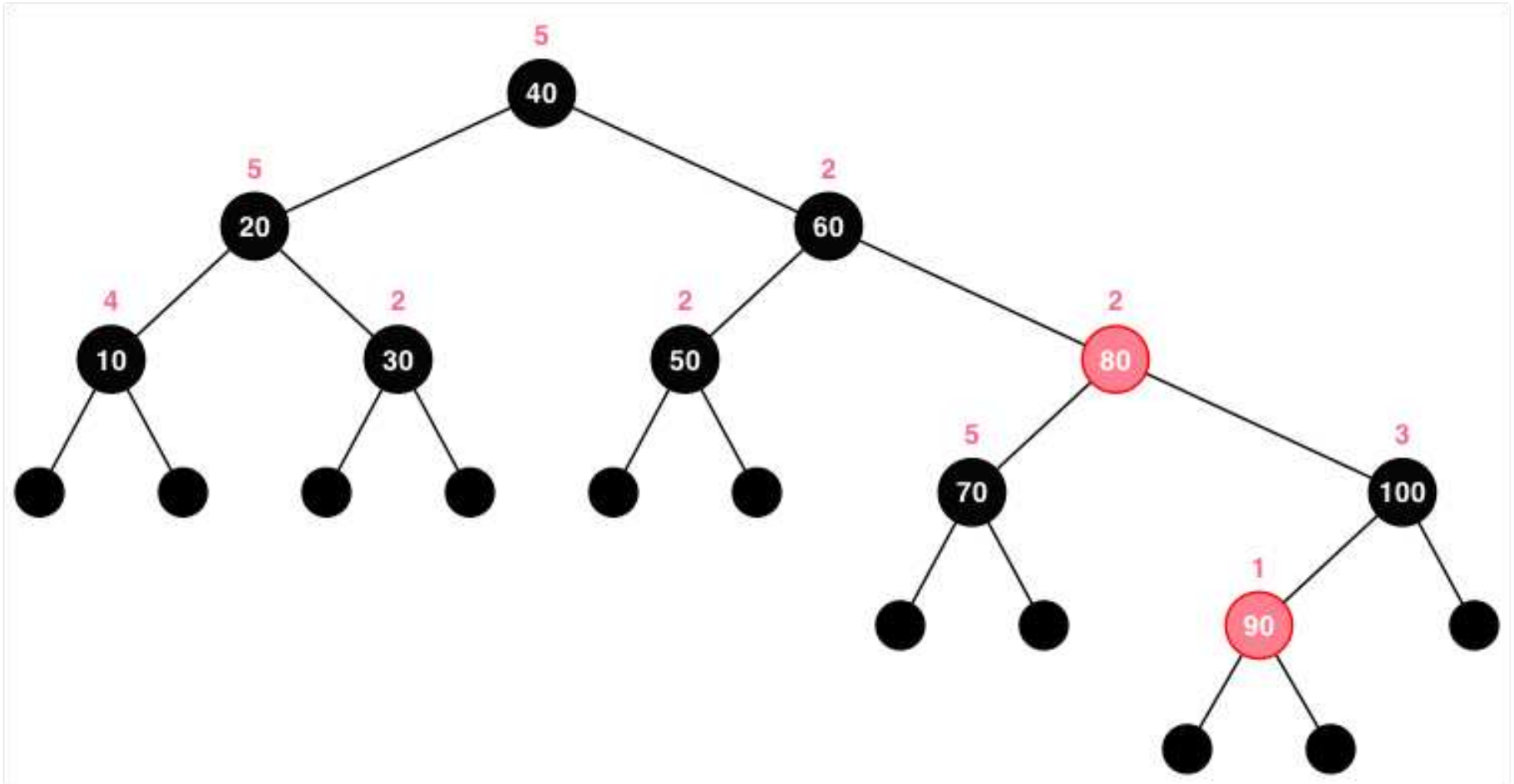
Les arbres rouges et noirs

Un **arbre rouge et noir** est un ABR tel que :

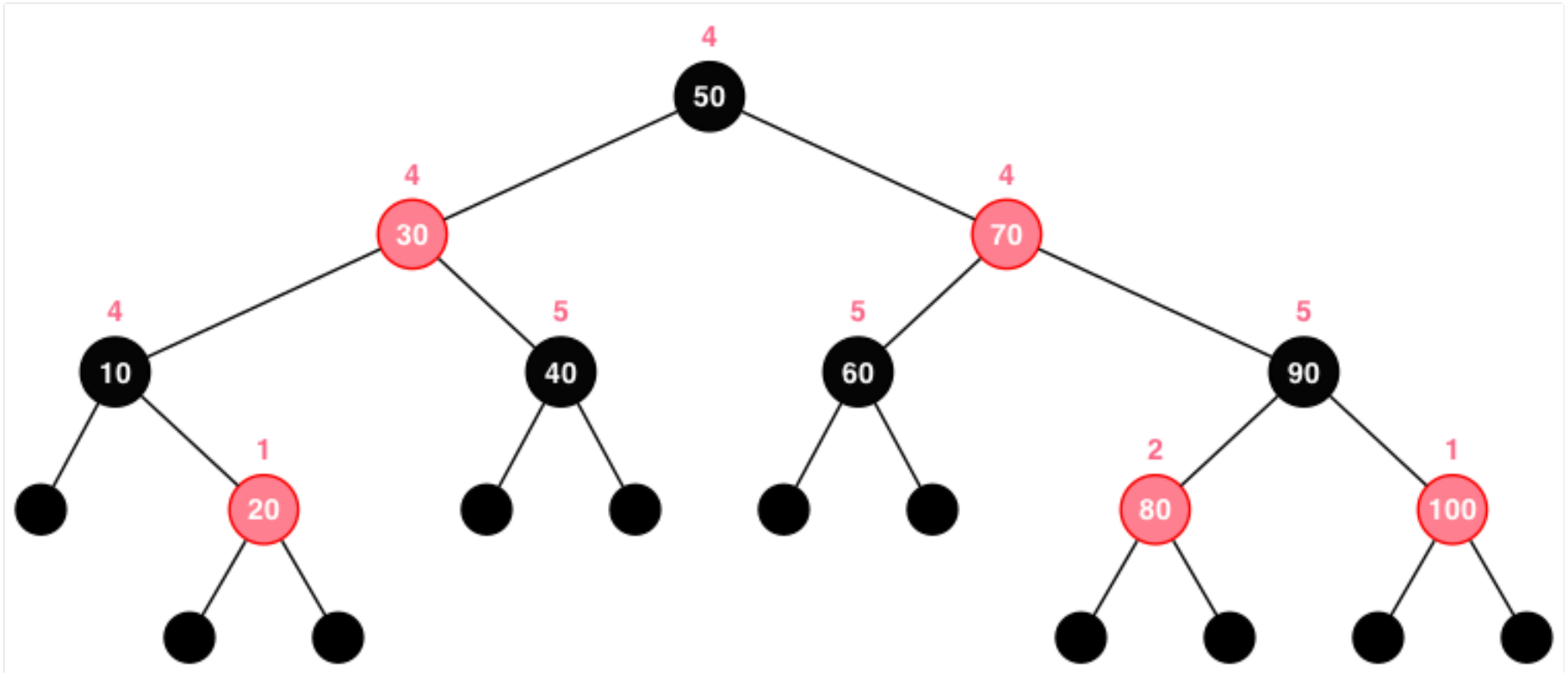
1. il est **complet** (pas de nœud d'arité 1),
2. seuls les nœuds internes portent des valeurs,
3. chaque nœud est soit rouge, soit noir,
4. la racine est noire,
5. les feuilles sont noires,
6. le père d'un nœud rouge est noir.
7. le nombre de nœuds noirs sur chaque branche est **constant**.

Hauteur noire = nombre de nœuds noirs sur chaque branche.

Arbres rouges et noirs : exemple 1



Arbres rouges et noirs : exemple 2



Plan

Les arbres rouges et noirs sont équilibrés

Preuve : sans récurrence. Soit t un arbre rouge et noir.

- $H =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre est complet et les branches ont au moins H nœuds.
  Donc tous les niveaux sont complets jusqu'à profondeur $H - 1$

Les arbres rouges et noirs sont équilibrés

Preuve : sans récurrence. Soit t un arbre rouge et noir.

- $H =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre est complet et les branches ont au moins H nœuds.

  Donc tous les niveaux sont complets jusqu'à profondeur $H - 1$
- Donc il y a au minimum $2^H - 1$ nœuds :

  $$\mathbf{2^H - 1 \leqslant n}, \text{ donc : } \mathbf{H \leqslant \log_2(n+1)}.$$

Les arbres rouges et noirs sont équilibrés

Preuve : sans récurrence. Soit t un arbre rouge et noir.

- $H =$ hauteur noire de t, $h =$ hauteur, $n =$ nombre de nœuds.
- L'arbre est complet et les branches ont au moins H nœuds.

  Donc tous les niveaux sont complets jusqu'à profondeur $H - 1$
- Donc il y a au minimum $2^H - 1$ nœuds :

  $$\boldsymbol{2^H - 1 \leqslant n}, \text{ donc : } \boldsymbol{H \leqslant \log_2(n+1).}$$
- Une branche a H nœuds noirs et au maximum $H - 1$ nœuds rouges : $\bullet - \bullet - \bullet - \bullet - \cdots - \bullet - \bullet - \bullet$.
- Donc $h \leqslant 2(H-1)$ et en utilisant l'inégalité ci-dessus :

  $$h \leqslant 2(\log_2(n+1) - 1).$$

Plan

Insertion dans un arbre rouge-noir

Un algorithme d'insertion dans un arbre rouge-noir :

- On insère **comme dans un ABR**,
- **Si création** d'un nouveau nœud,
  - il remplace l'une des anciennes feuilles noires,
  - on le colore en **rouge**,
  - ses deux fils sont des feuilles (noires).

- **Problème potentiel**

Insertion dans un arbre rouge-noir

Un algorithme d'insertion dans un arbre rouge-noir :

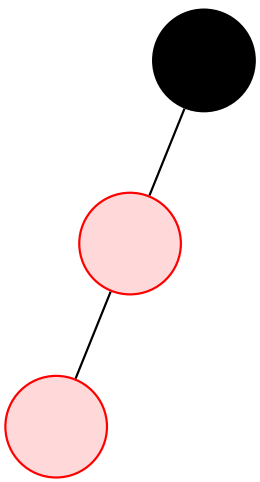
- On insère **comme dans un ABR**,
- **Si création** d'un nouveau nœud,
  - il remplace l'une des anciennes feuilles noires,
  - on le colore en **rouge**,
  - ses deux fils sont des feuilles (noires).

- **Problème potentiel**
  - l'arbre obtenu peut avoir **2 nœuds rouges père-fils**.

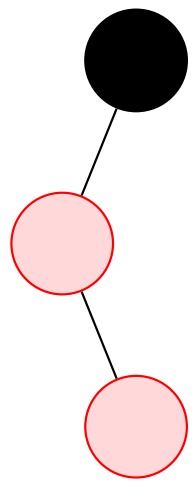
Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

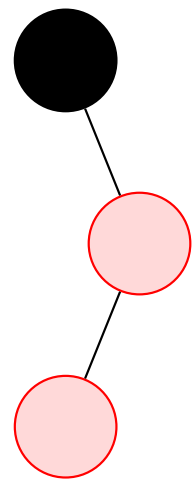
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



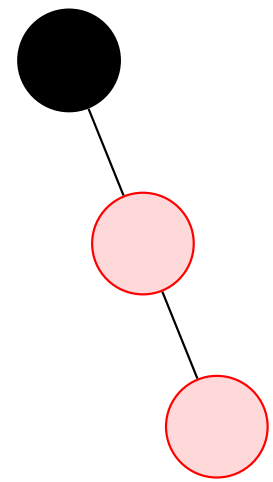
Cas 1a



Cas 2a



Cas 2b

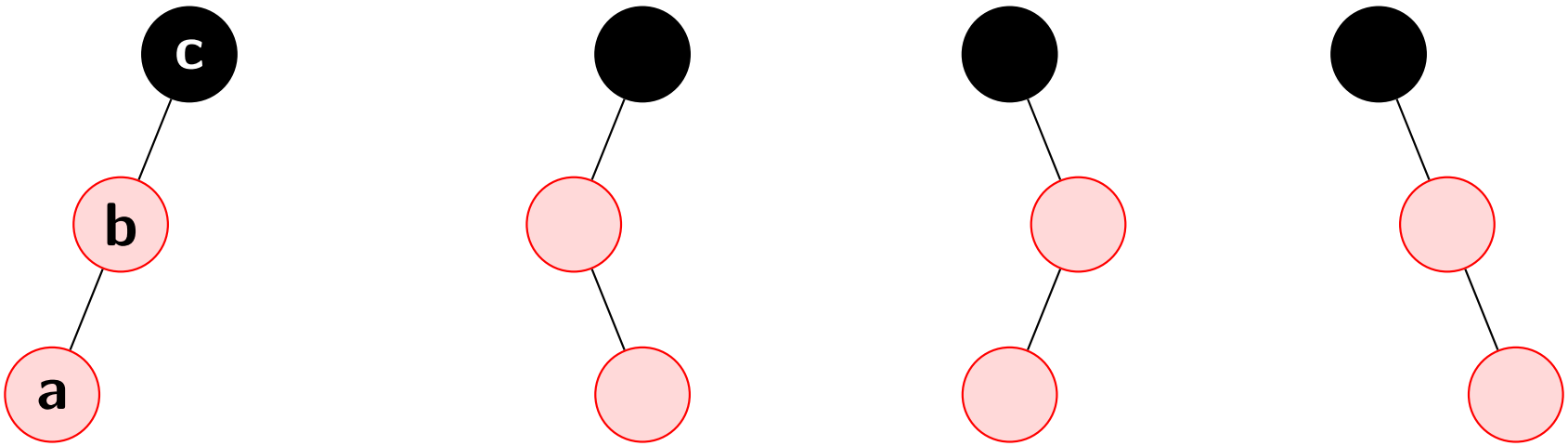


Cas 1b

Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:

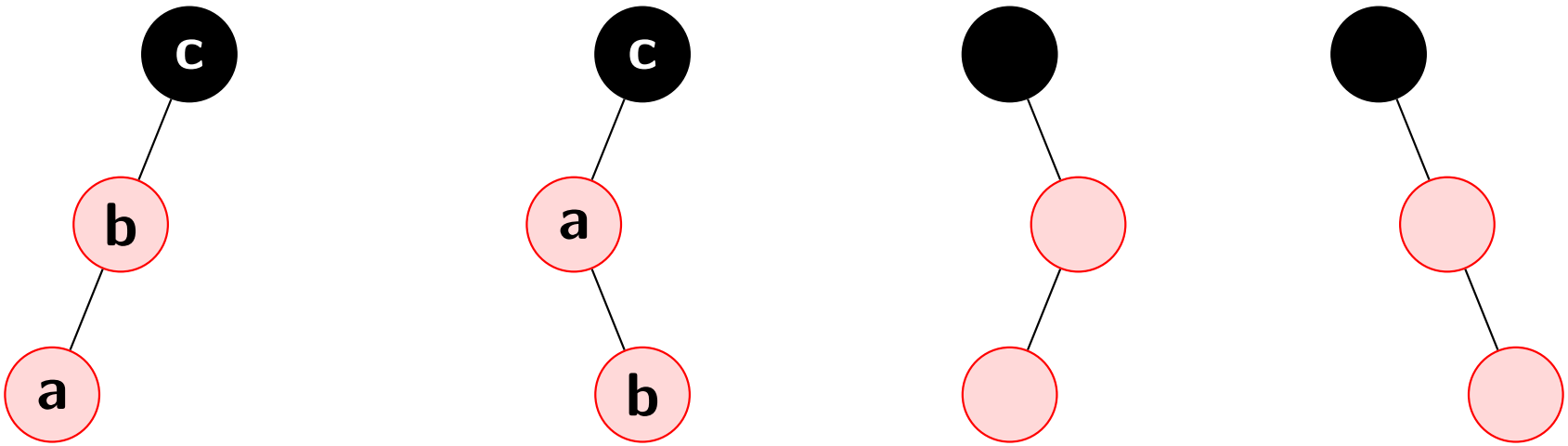


Cas 1a **Cas 2a** **Cas 2b** **Cas 1b**

Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:

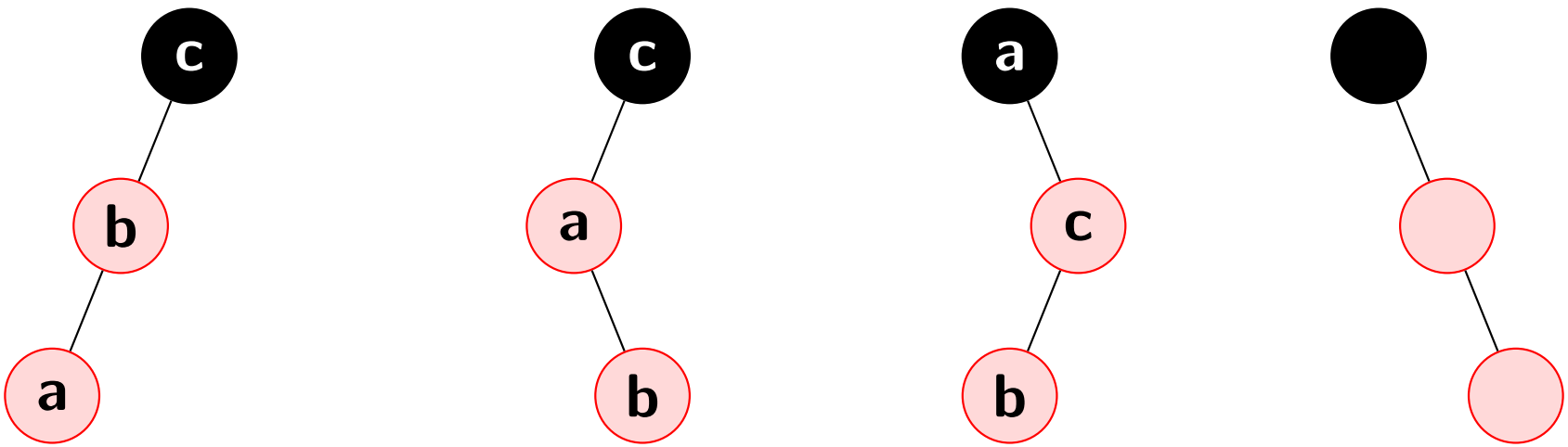


Cas 1a **Cas 2a** **Cas 2b** **Cas 1b**

Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:

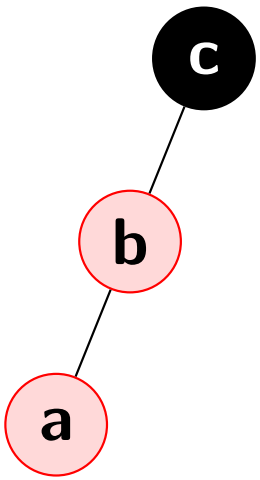


Cas 1a **Cas 2a** **Cas 2b** **Cas 1b**

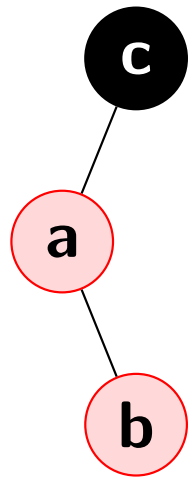
Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

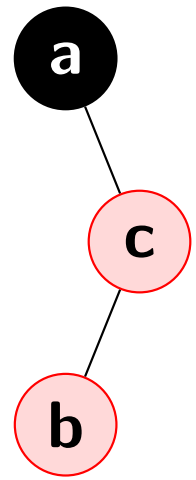
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



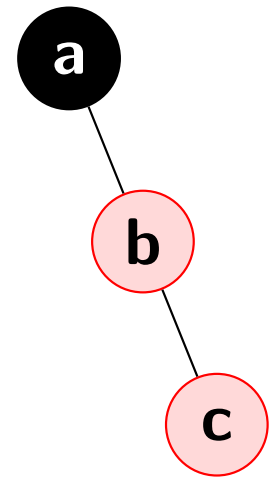
Cas 1a



Cas 2a



Cas 2b

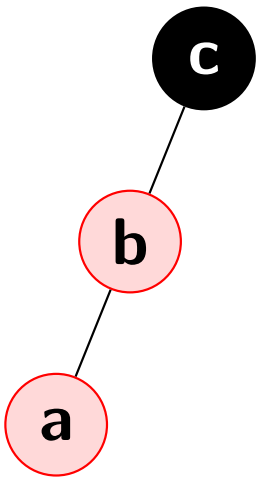


Cas 1b

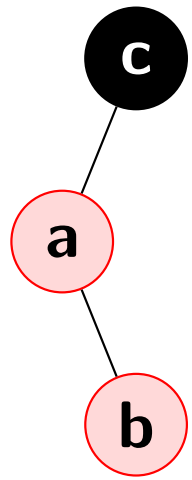
Insertion dans un arbre rouge-noir

Réorganisation de l'arbre pour

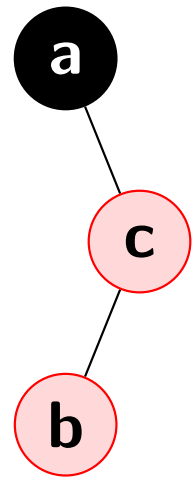
- éviter 2 nœuds rouges consécutifs.
- conserver les autres propriétés des arbres rouges et noirs.
- Après insertion : 4 cas possibles avec $\mathbf{a} < \mathbf{b} < \mathbf{c}$:



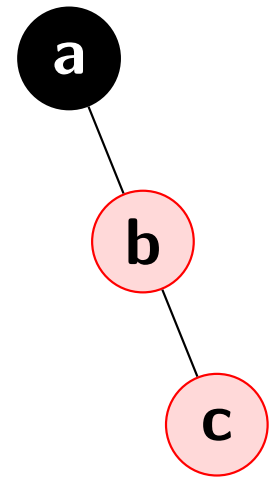
Cas 1a



Cas 2a



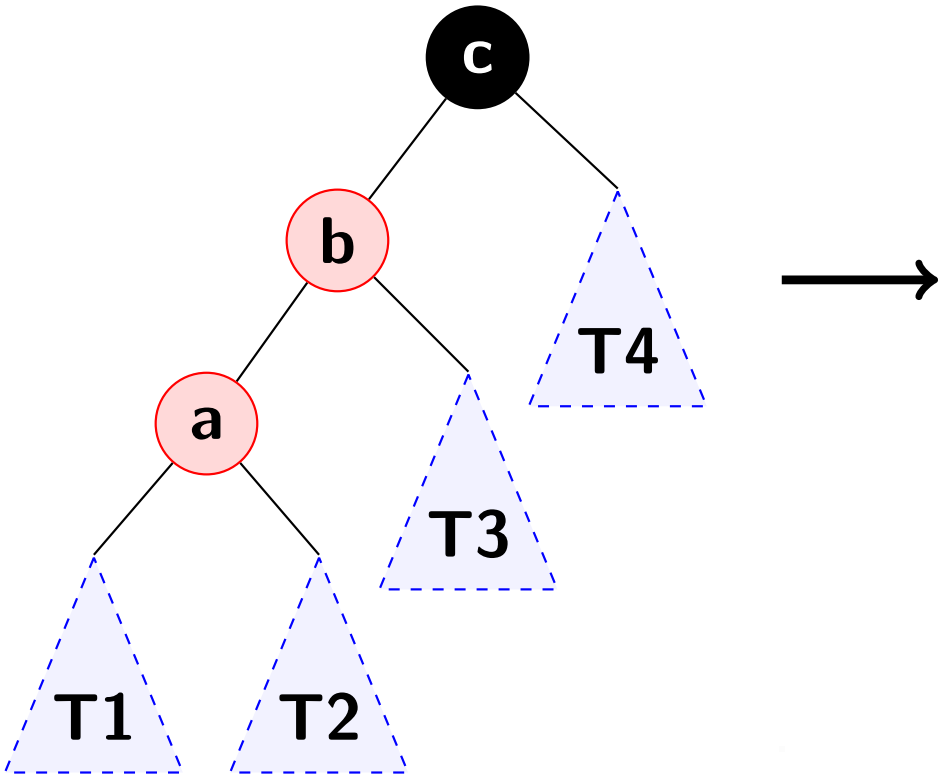
Cas 2b



Cas 1b

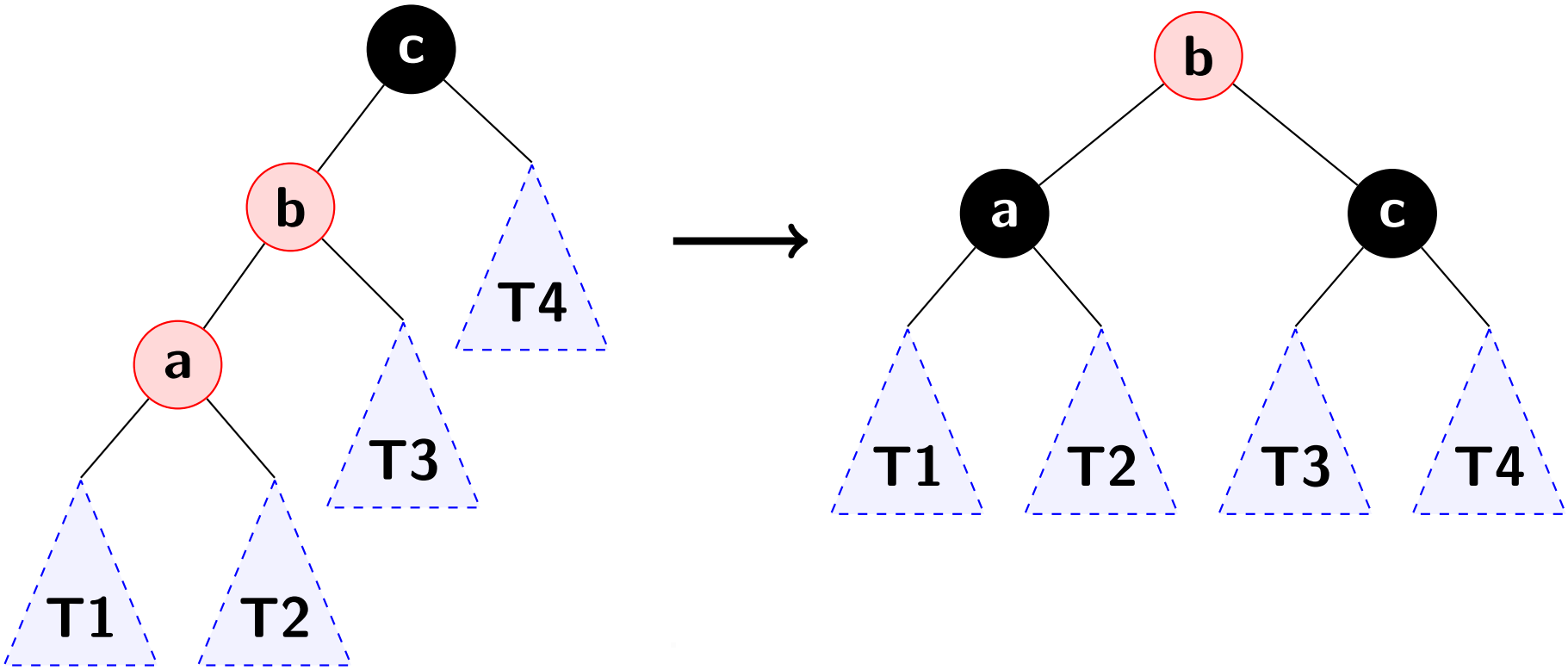
On applique une **transformation** dans chacun des 4 cas.

Insertion dans un arbre rouge-noir : cas 1



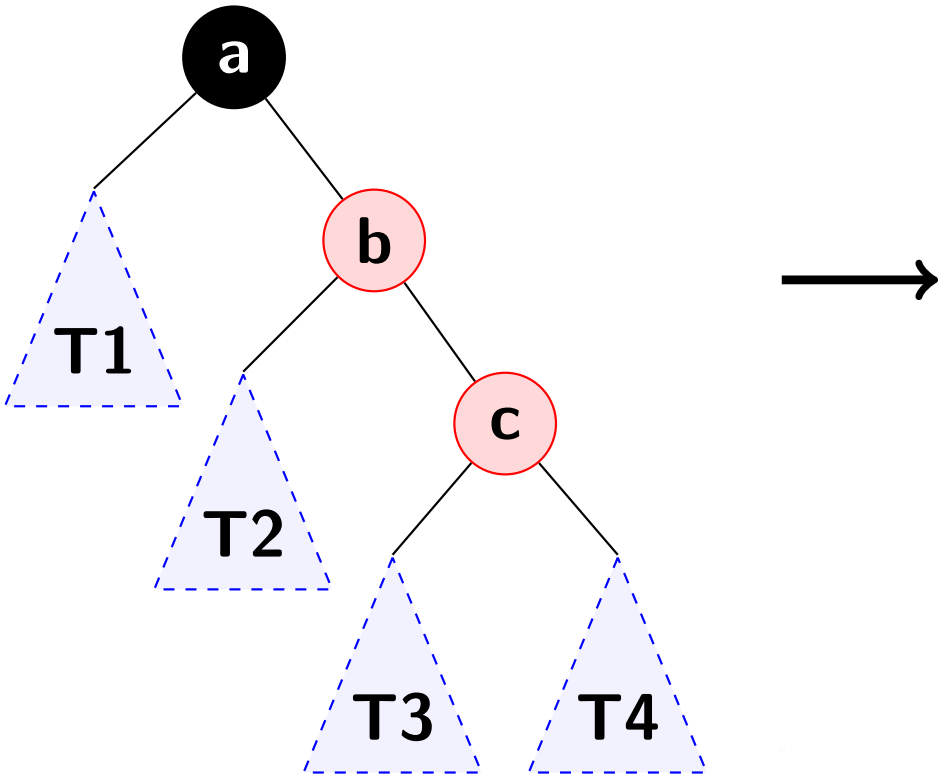
$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 1



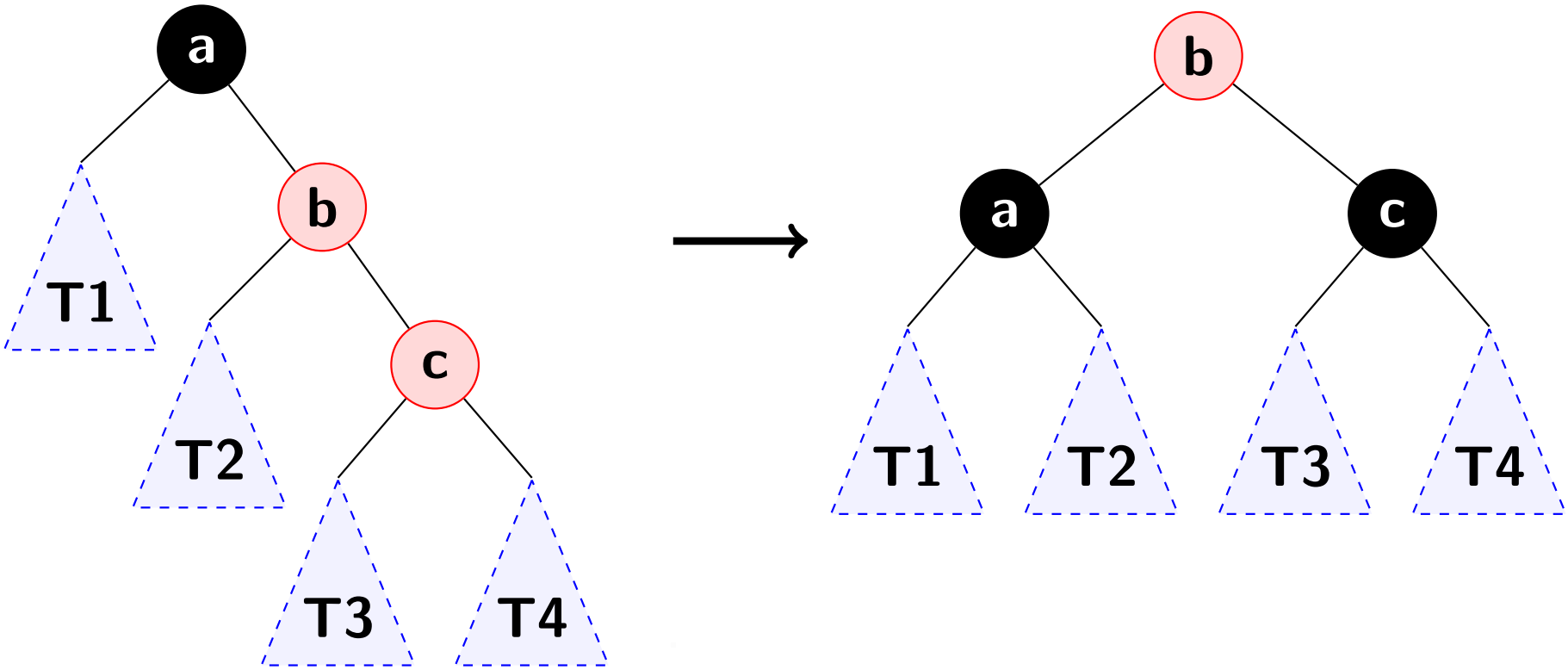
$$\mathbf{cles(T1)} < \mathbf{a} < \mathbf{cles(T2)} < \mathbf{b} < \mathbf{cles(T3)} < \mathbf{c} < \mathbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 1b



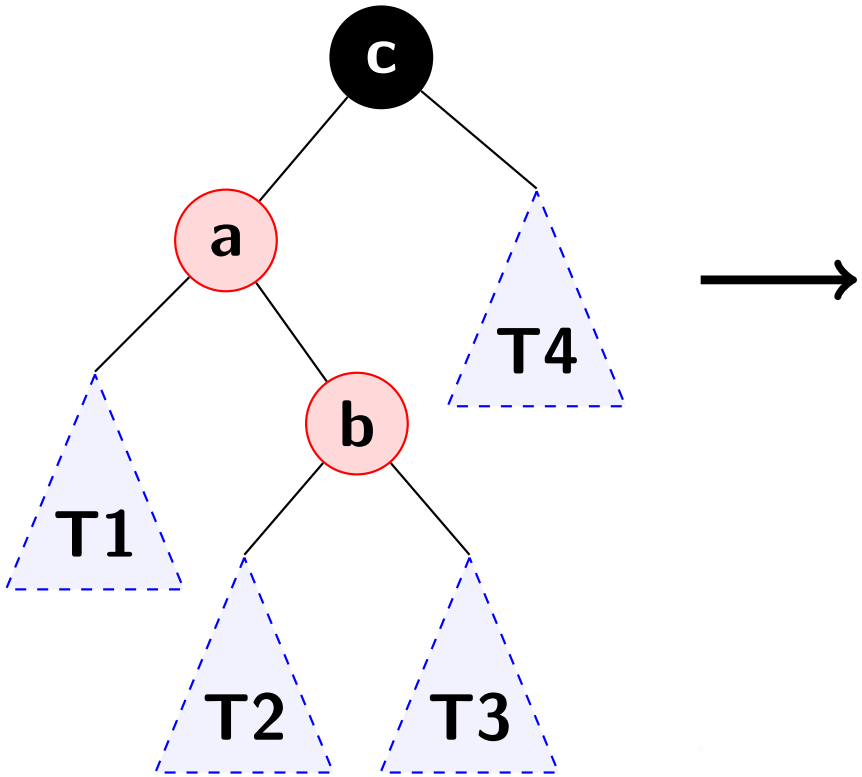
$$\mathbf{cles(T1)} < \mathbf{a} < \mathbf{cles(T2)} < \mathbf{b} < \mathbf{cles(T3)} < \mathbf{c} < \mathbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 1b



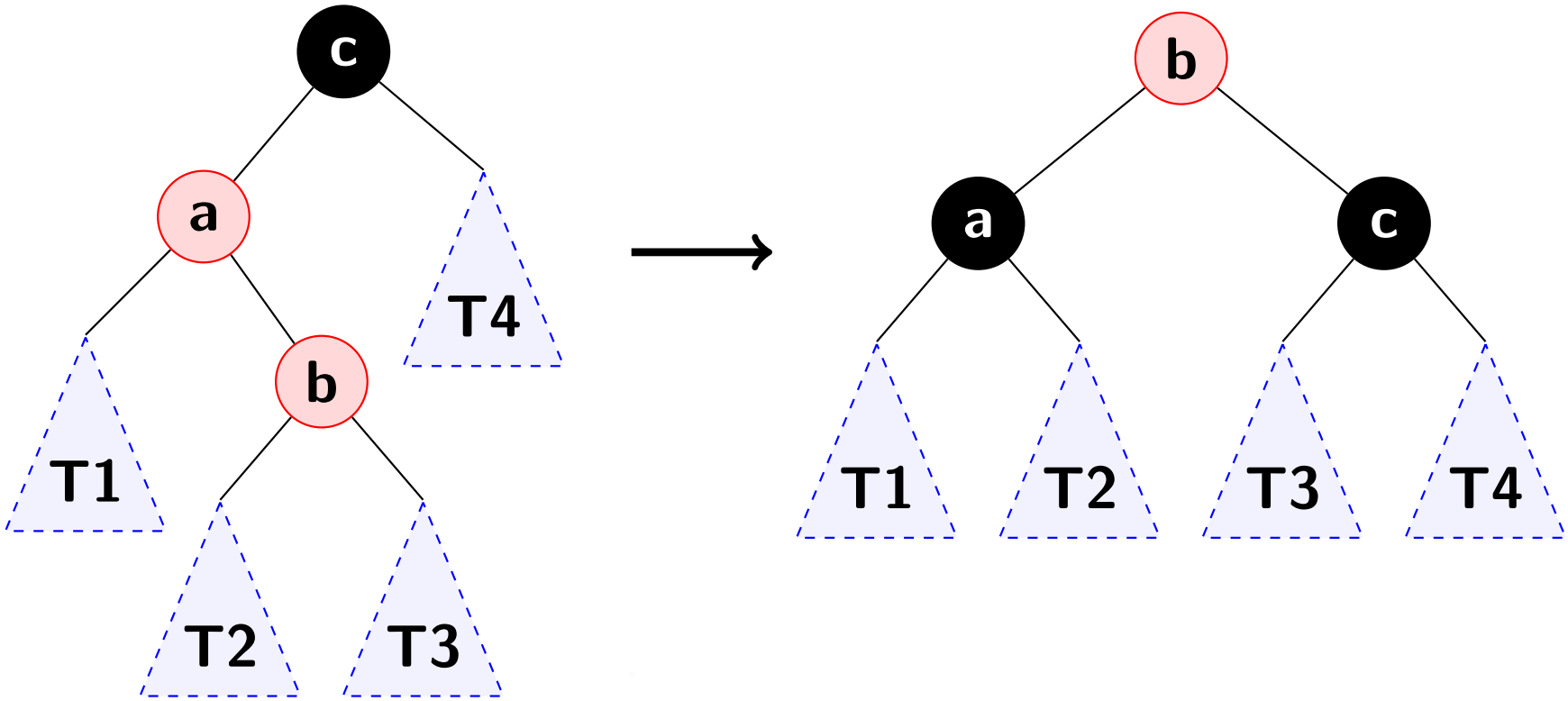
$$\mathbf{cles(T1) < a < cles(T2) < b < cles(T3) < c < cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 2a



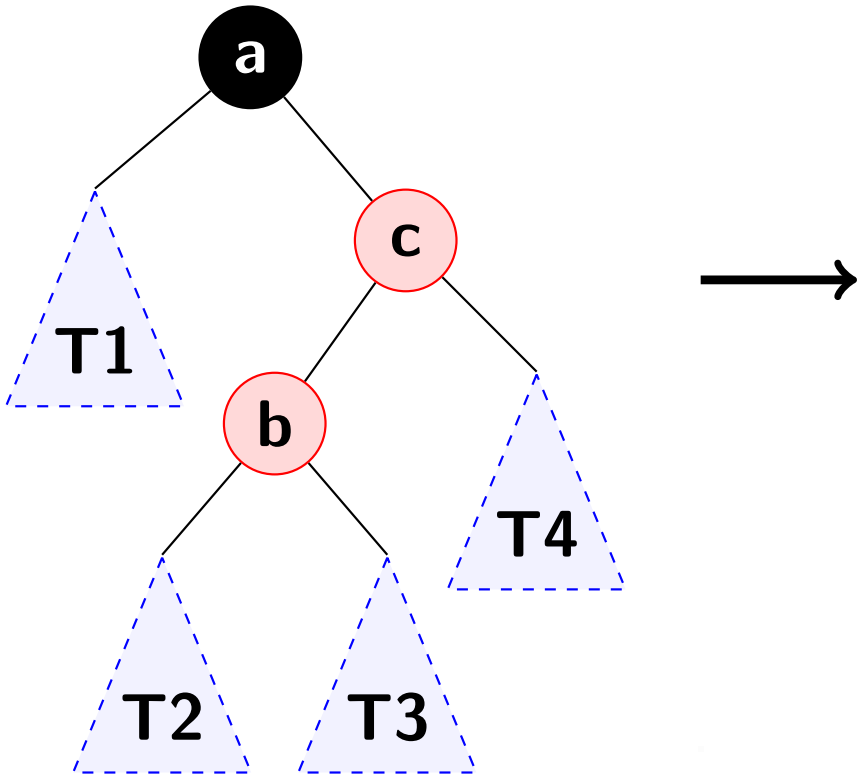
$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 2a



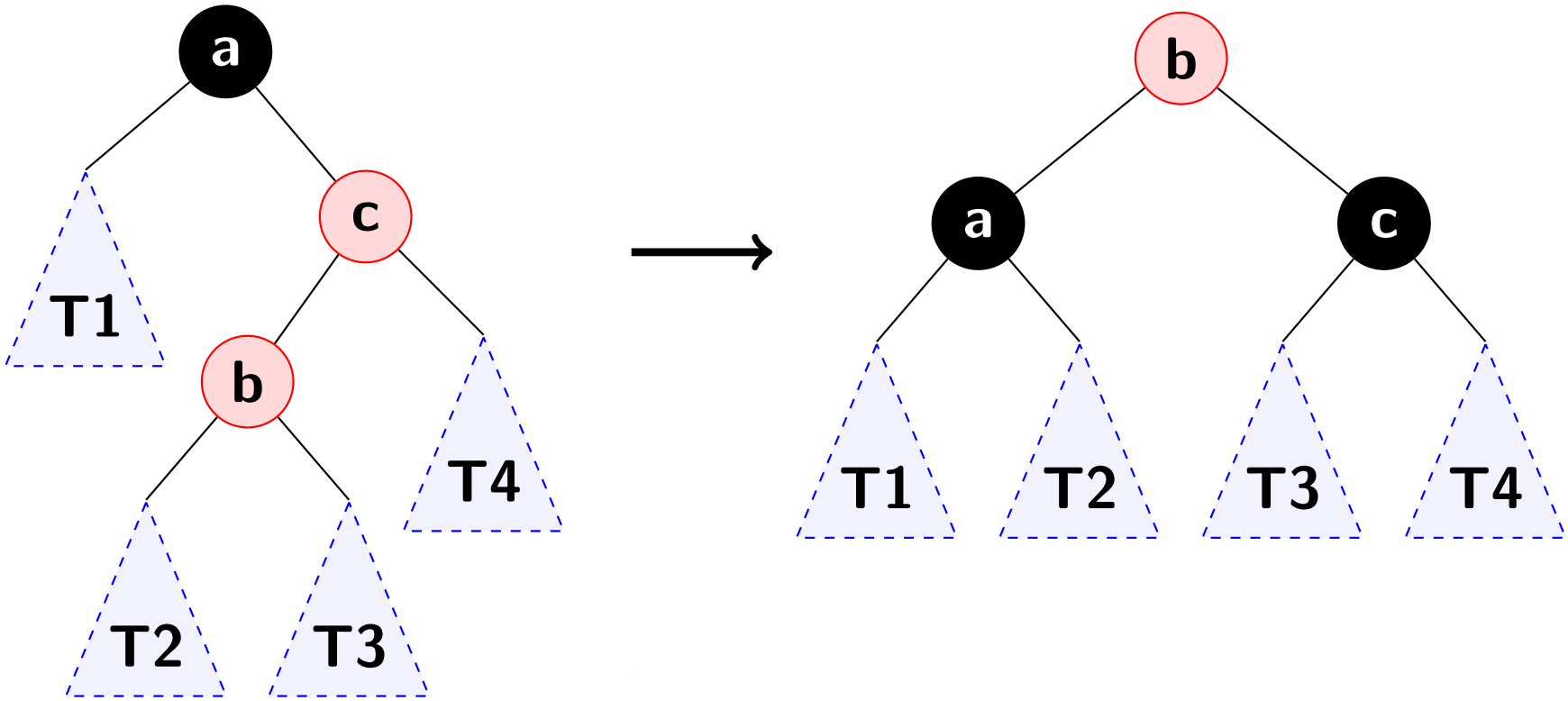
$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

Insertion dans un arbre rouge-noir : cas 2b



$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

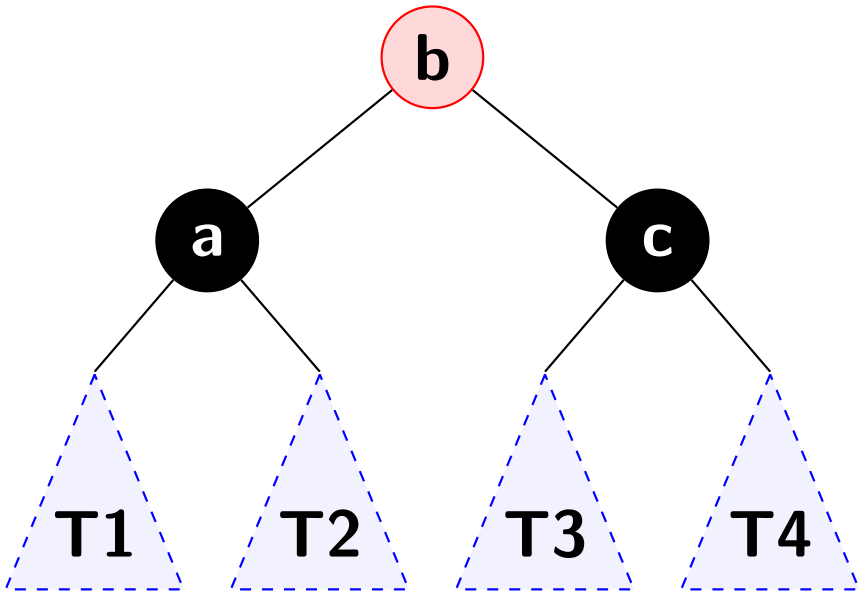
Insertion dans un arbre rouge-noir : cas 2b



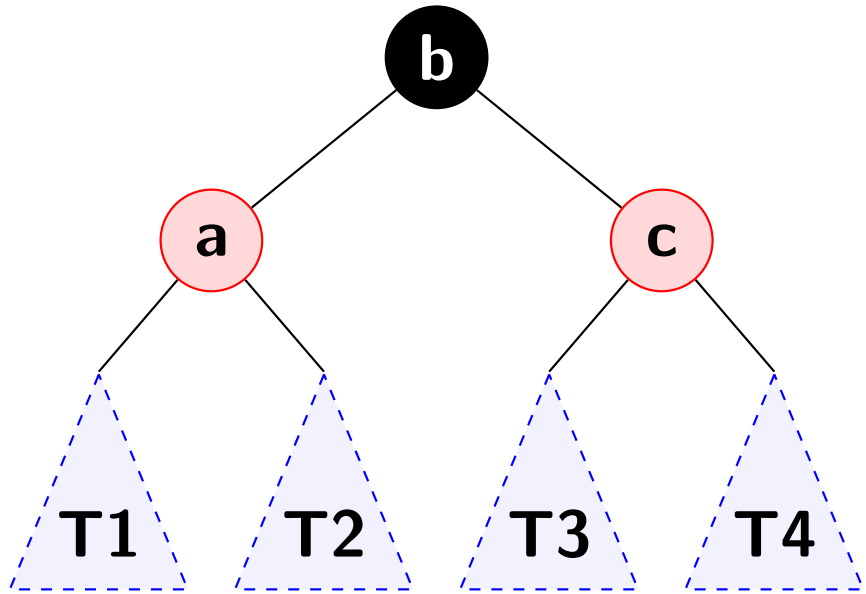
$$\textbf{cles(T1)} < \textbf{a} < \textbf{cles(T2)} < \textbf{b} < \textbf{cles(T3)} < \textbf{c} < \textbf{cles(T4)}$$

Question...

Au lieu de



Peut-on choisir



? ? ? ?

Insertion et rééquilibrage

- La transformation crée un nœud rouge à la racine du sous-arbre sur lequel on travaille.
- $\Rightarrow$ Risque de créer à nouveau 2 nœuds rouges consécutifs.
- $\Rightarrow$ On doit ré-équilibrer récursivement.
- Si on arrive à la racine, on peut la colorier en noir.

  Augmente la hauteur noire de 1 sans changer le fait qu'elle est constante.

Suppression dans les arbres rouges-noirs

- Plus compliquée.
- On supprime d'abord comme dans un ABR normal.
- Problème quand on supprime un nœud noir : le nombre de nœuds noirs est maintenant inférieur.
- **Une solution** :
  - créer un nœud « double noir »,
  - le faire remonter, potentiellement jusqu'à la racine, par rotations.
  - le transformer en nœud noir s'il est à la racine.
  - la remontée du nœud double noir peut conduire à créer un nœud noir négatif.

Arbres rouges et noirs : complexité insertion, suppression

- La phase d'insertion sans ré-équilibrage prend un temps $O(\log(n))$
- Chaque ré-équilibrage local demande un temps $O(1)$
- Au pire $O(h) = O(\log(n))$ rééquilibrages chacuns coûtant $O(1)$ pour la suppression.
- **En tout** : $O(\log(n))$.

Plan

Les AVL

Les AVL sont une autre sorte d'arbres binaires de recherche équilibrés.

- **Équilibre** d'un nœud =

  hauteur(sous-arbre gauche) − hauteur(sous-arbre droit).
- **AVL** = ABR dont chaque nœud a un équilibre $-1, 0$ ou 1.

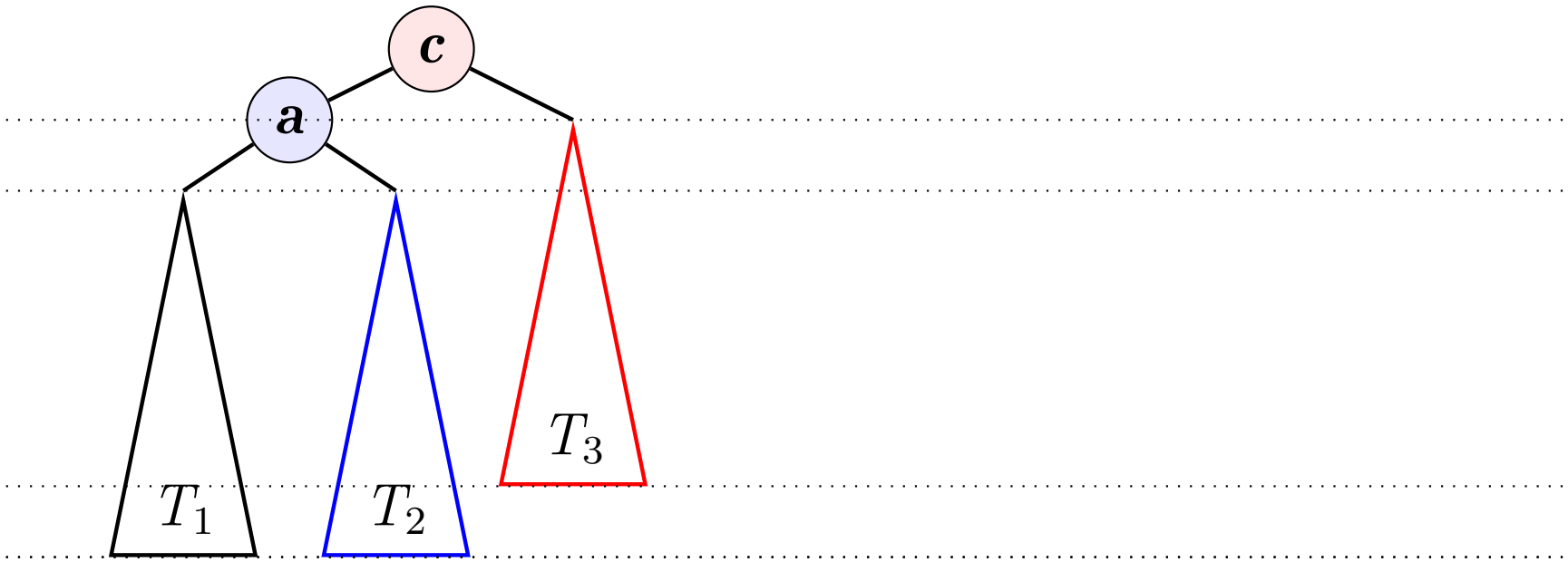
Insertion

- Insertion : on commence par insérer comme dans un **ABR**.
- Si création d'un nœud x, l'arbre n'est peut-être plus un **AVL**.
  Un des nœuds a pu passer d'équilibre ± 1 à ± 2.
- $c = 1^{\text{er}}$ nœud sur la branche de x à la racine d'équilibre ± 2.
- On doit **réparer** l'AVL pour rétablir les propriétés.

AVL : équilibrage, cas 1a

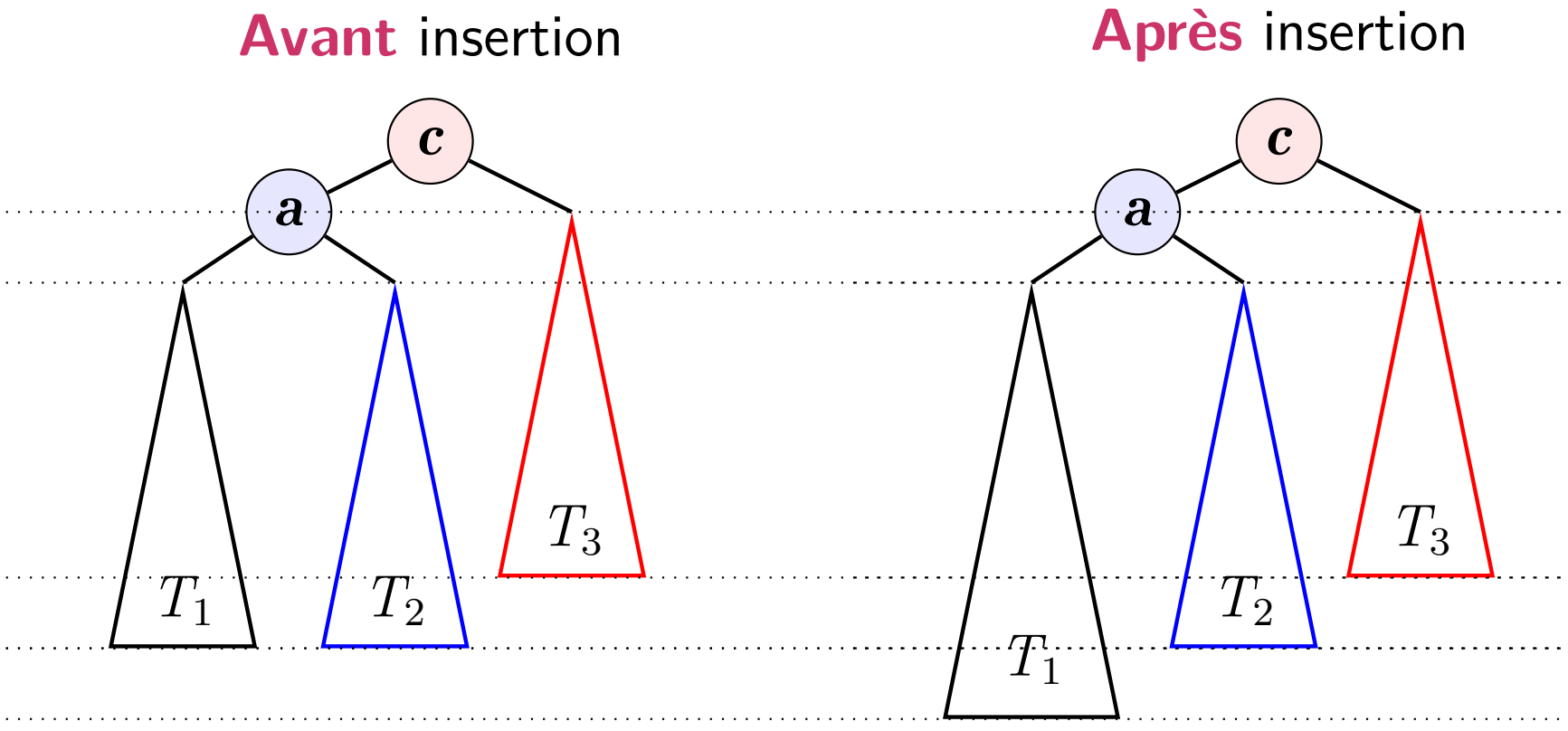
- Insertion dans le sous-arbre gauche du sous-arbre gauche (GG).

Avant insertion



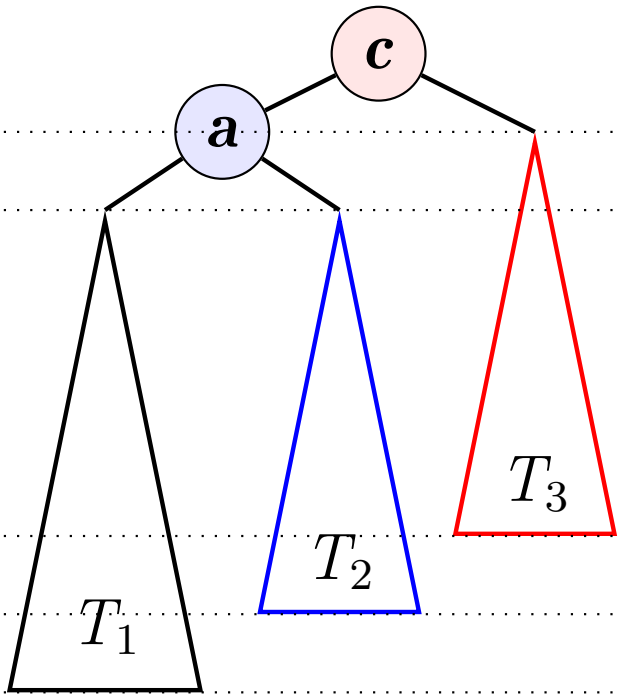
AVL : équilibrage, cas 1a

- Insertion dans le sous-arbre gauche du sous-arbre gauche (GG).



AVL : équilibrage, cas 1a

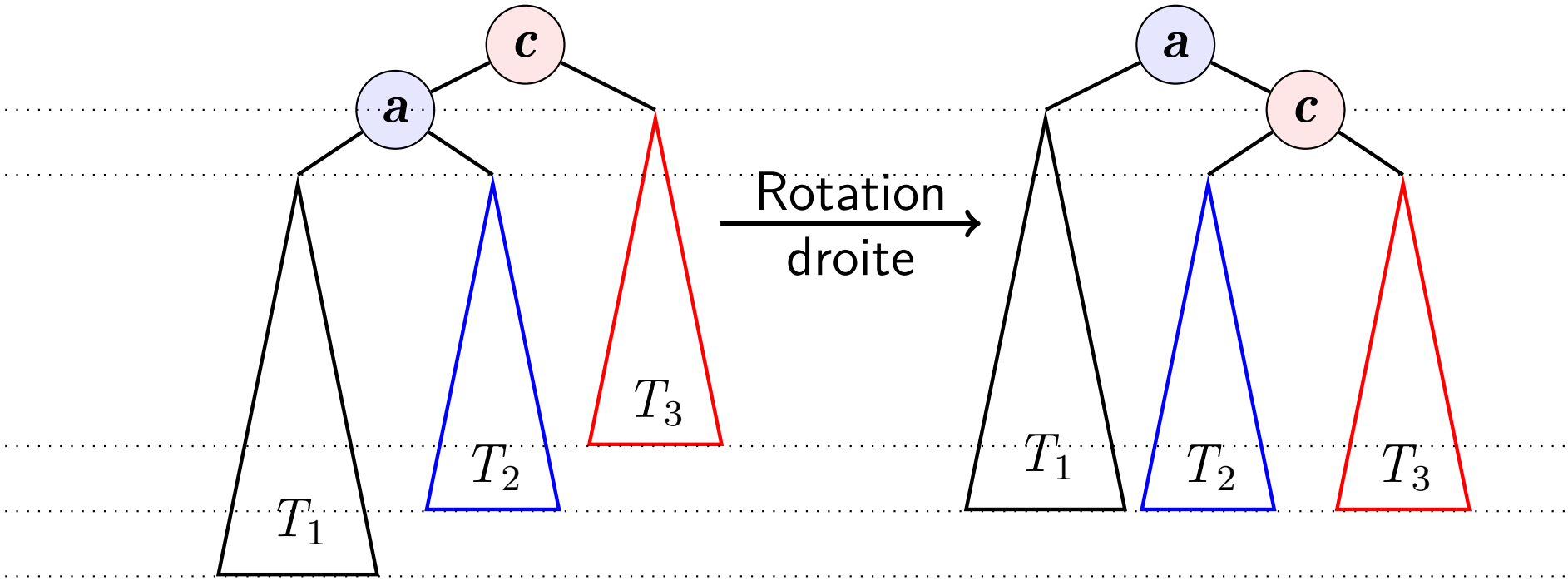
- hauteur(GG) = hauteur(D) + 1



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < c < \textbf{clés}(T_3)$$

AVL : équilibrage, cas 1a

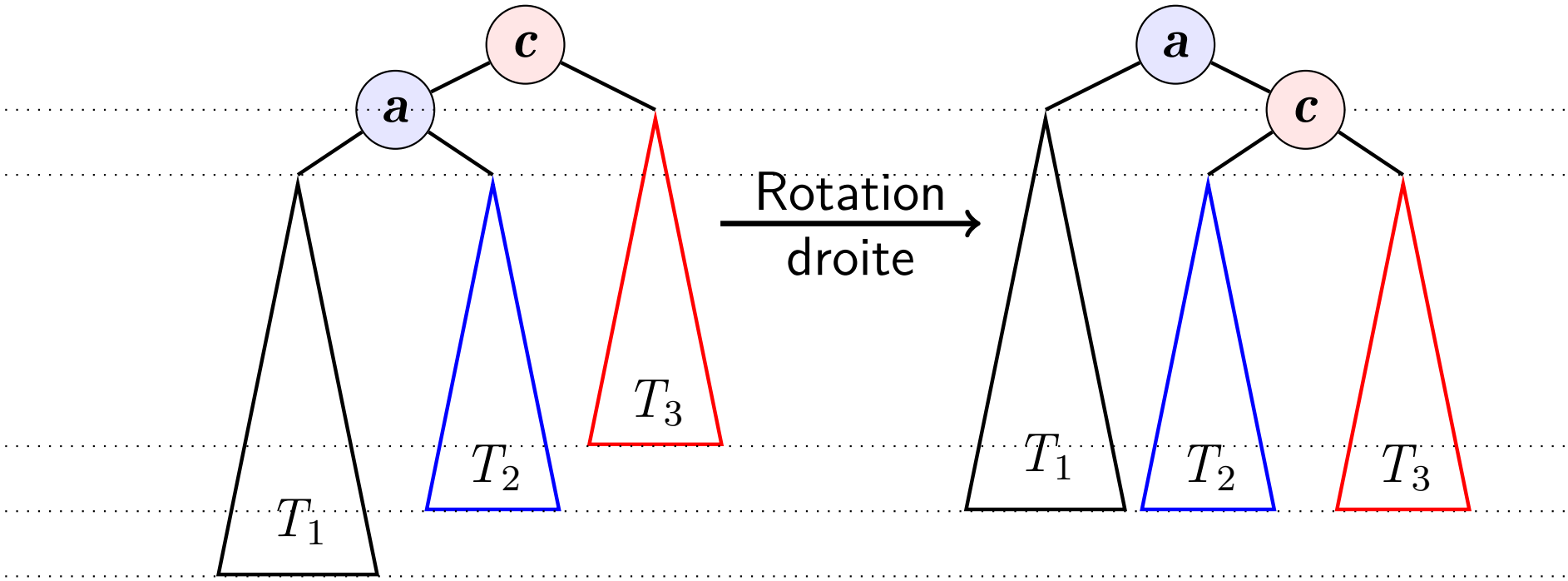
- hauteur(GG) = hauteur(D) + 1



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < c < \textbf{clés}(T_3)$$

AVL : équilibrage, cas 1a

- hauteur(GG) = hauteur(D) + 1



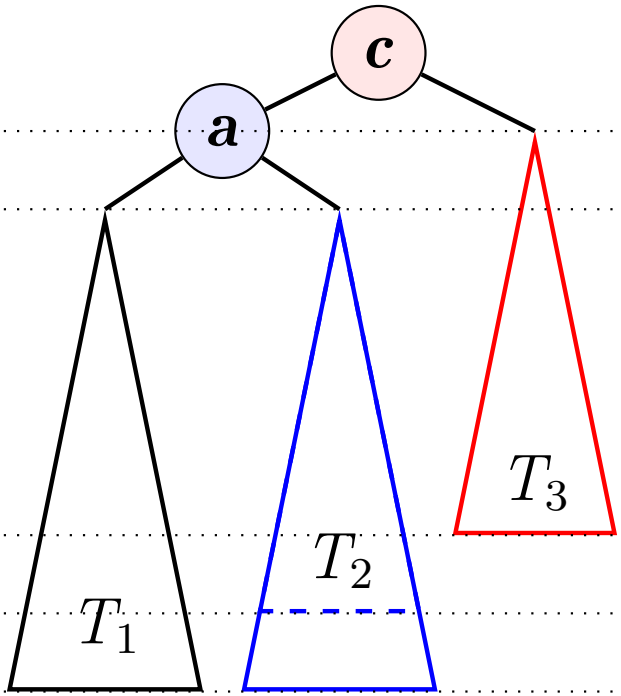
$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < c < \textbf{clés}(T_3)$$

On retrouve la hauteur avant insertion $\rightsquigarrow$ terminé !

AVL : équilibrage, cas 1a

- **Remarque** Fonctionne même si T_2 est plus haut.

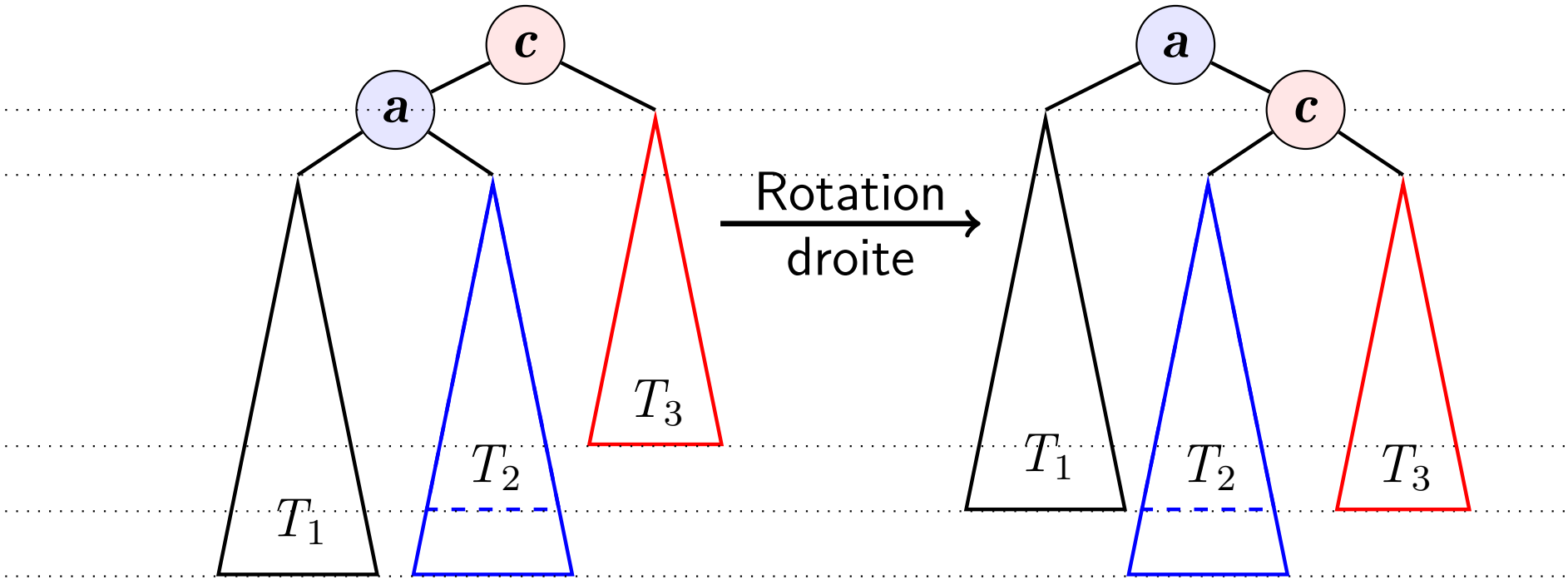
  Utile dans le cas de la suppression.



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < c < \textbf{clés}(T_3)$$

AVL : équilibrage, cas 1a

- **Remarque** Fonctionne même si T_2 est plus haut.
  Utile dans le cas de la suppression.



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < c < \textbf{clés}(T_3)$$

AVL : équilibrage, cas 1b

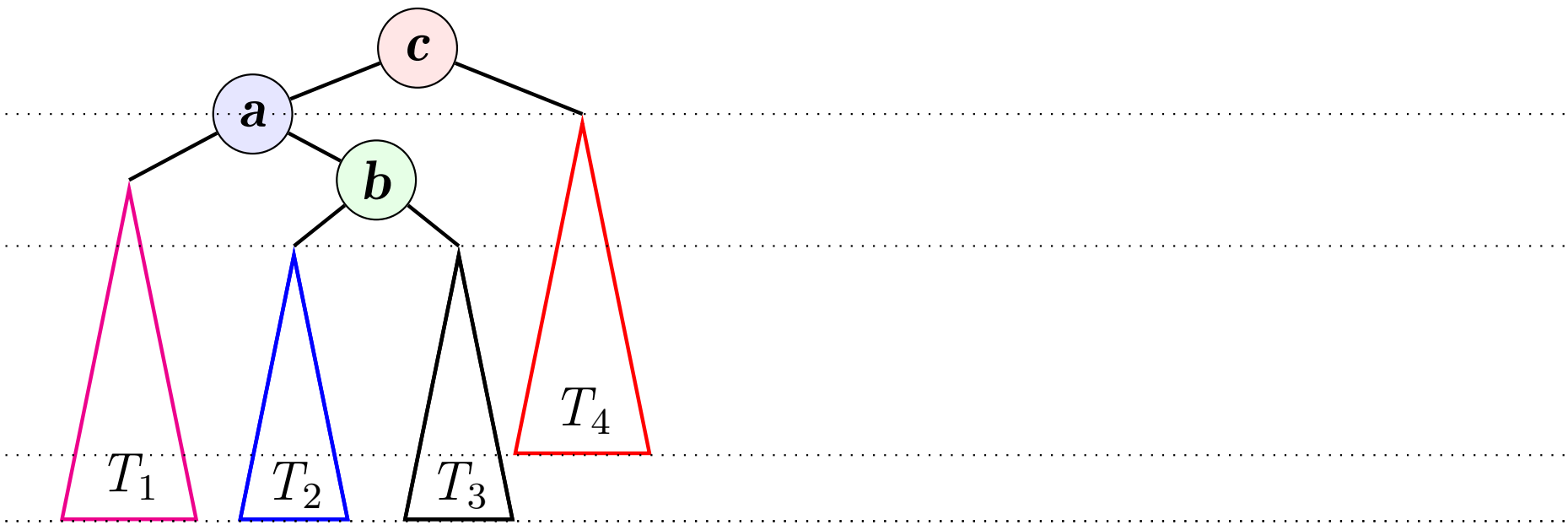
- $\text{hauteur}(\text{DD}) = \text{hauteur}(\text{G}) + 1$

Symétrique du cas 1a) : rotation gauche.

AVL : équilibrage, cas 2a

- Insertion dans le sous-arbre droit du sous-arbre gauche (GD).
  La hauteur de T_2 ou celle de T_3 a augmenté.

Avant insertion

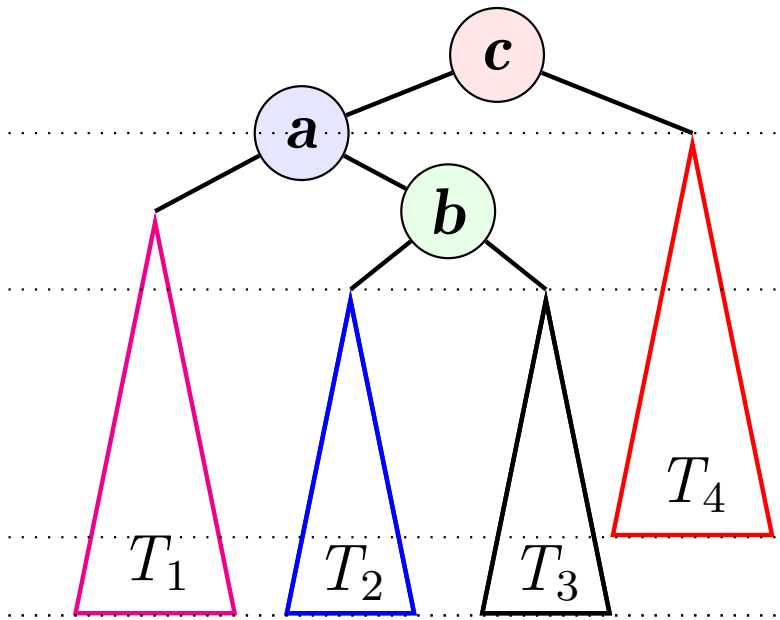


$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

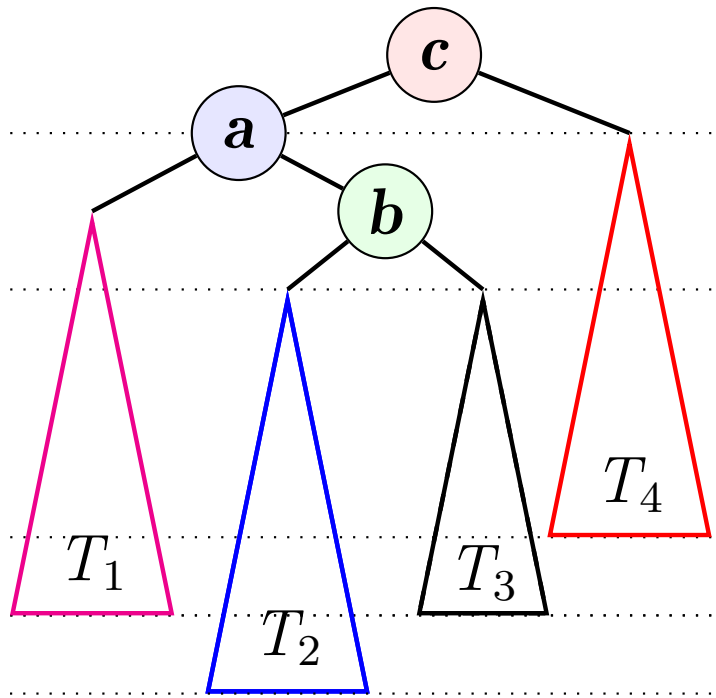
AVL : équilibrage, cas 2a

- Insertion dans le sous-arbre droit du sous-arbre gauche (GD).
  La hauteur de T_2 ou celle de T_3 a augmenté.

Avant insertion



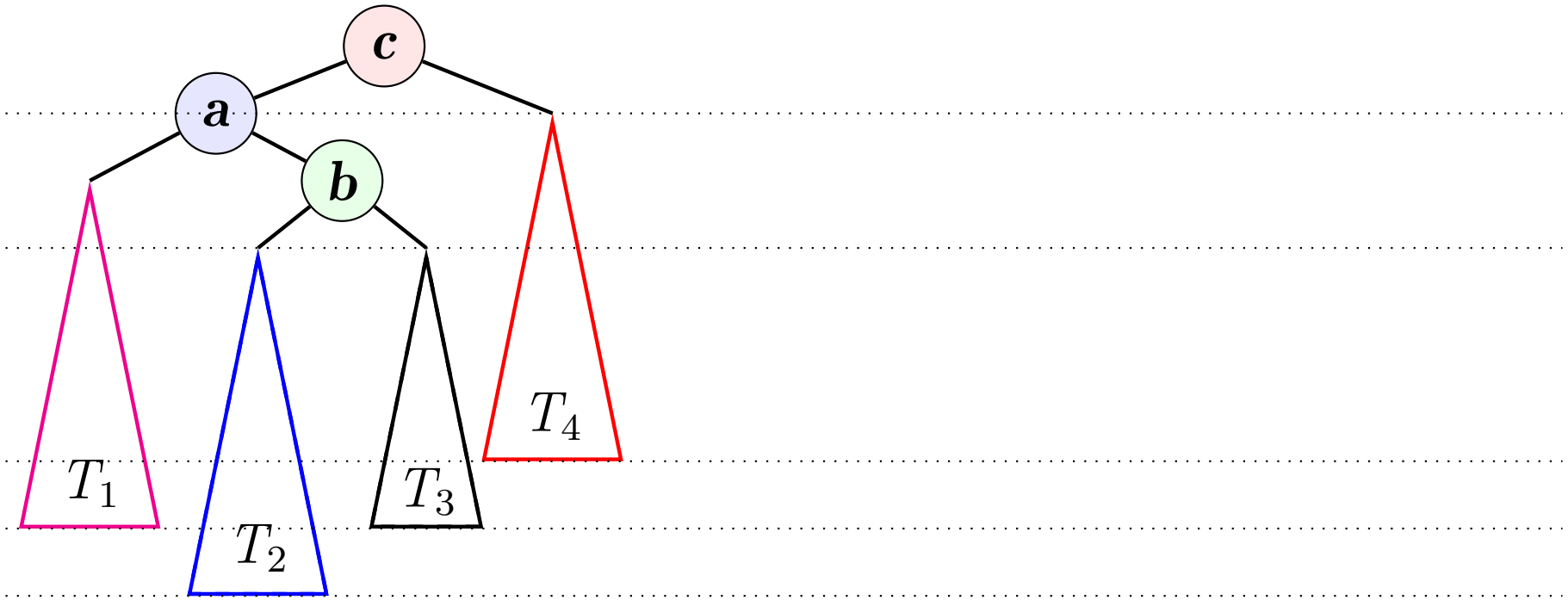
Après insertion



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

AVL : équilibrage, cas 2a

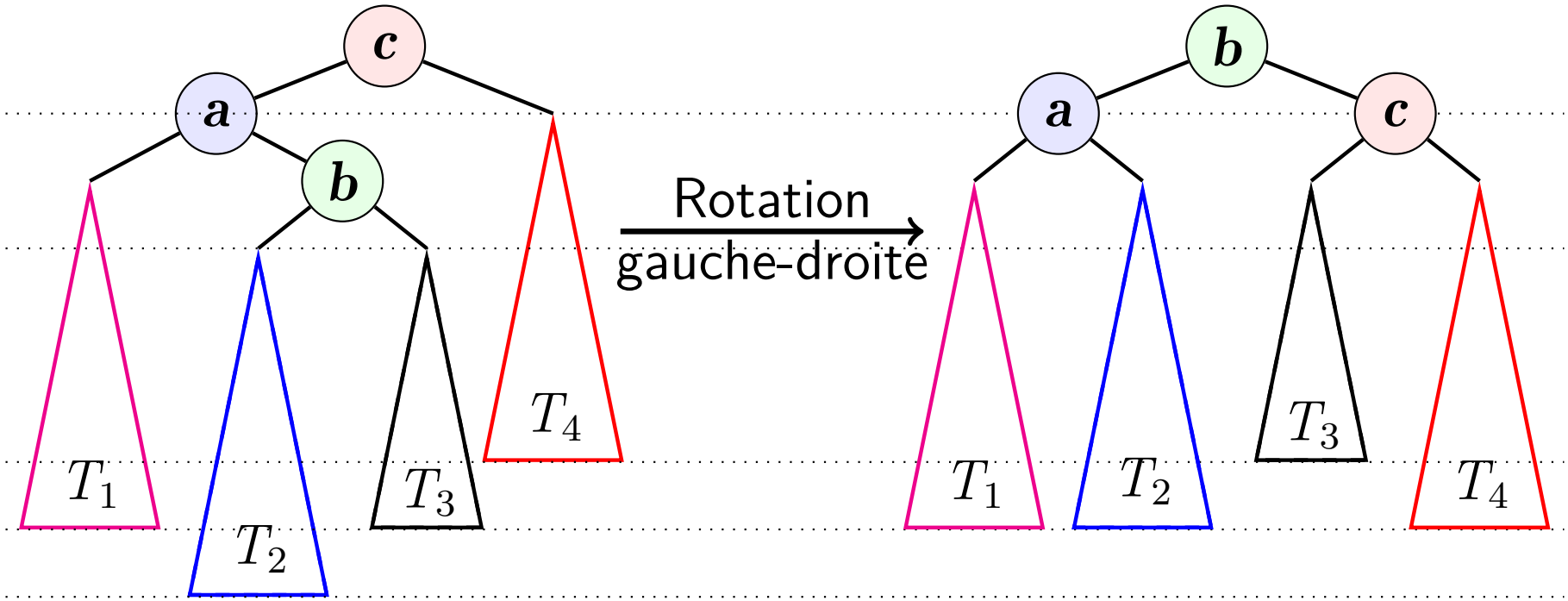
- hauteur(GD) = hauteur(D) + 1



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

AVL : équilibrage, cas 2a

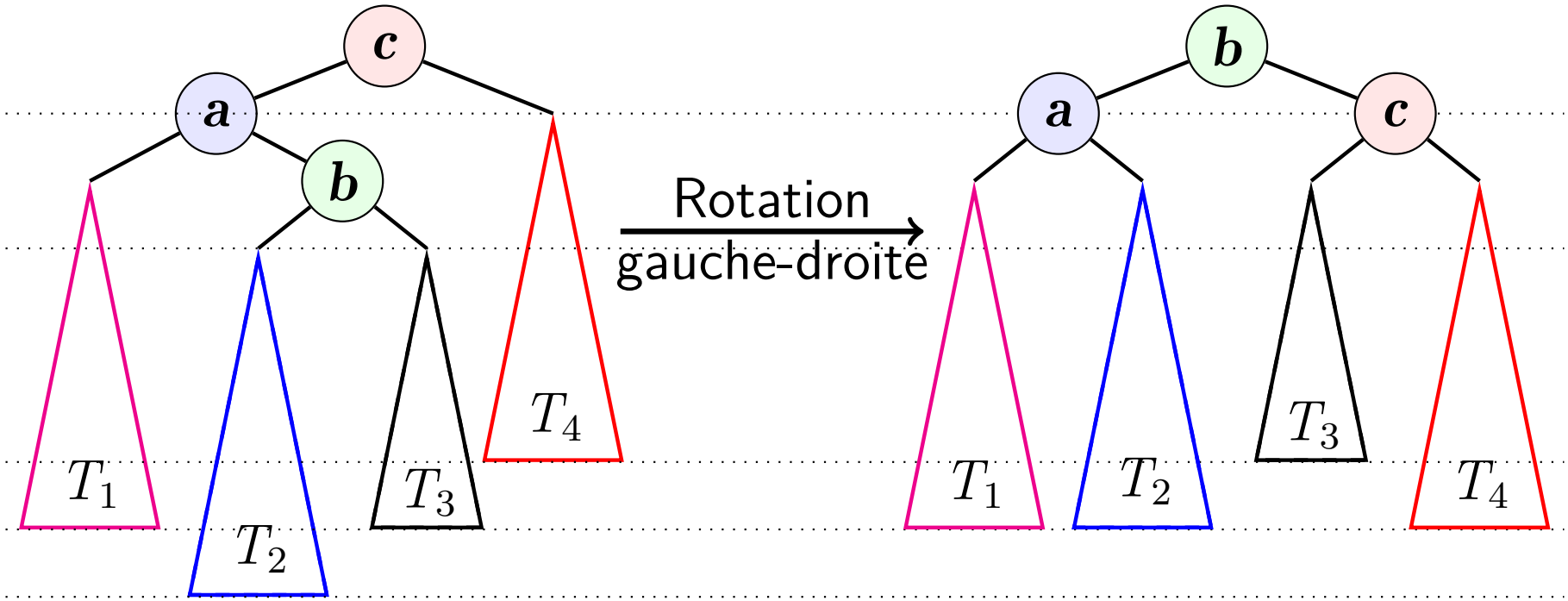
- hauteur(GD) = hauteur(D) + 1



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

AVL : équilibrage, cas 2a

- hauteur(GD) = hauteur(D) + 1



$$\textbf{clés}(T_1) < a < \textbf{clés}(T_2) < b < \textbf{clés}(T_3) < c < \textbf{clés}(T_4)$$

On retrouve la hauteur avant insertion $\rightsquigarrow$ terminé !

AVL : équilibrage, cas 2b

- $\text{hauteur}(\text{DG}) = \text{hauteur}(\text{D}) + 1$

Symétrique du cas 2a) : rotation droite-gauche.

Suppression dans les arbres AVL

- La suppression se gère de la même façon que l'insertion.
  - Suppression comme dans les arbres binaires de recherche.
  - Ré-équilibrage pour retrouver la propriété "AVL".
- Un ré-équilibrage peut déséquilibrer un nœud plus haut

$\rightsquigarrow$ peut nécessiter plusieurs équilibrages, au pire jusqu'à la racine.

Arbres AVL : complexité insertion, suppression

- La phase d'insertion prend un temps $O(\log(n))$
- Chaque ré-équilibrage local demande un temps $O(1)$
  - 1 rééquilibrage pour l'insertion.
  - Au pire $O(h) = O(\log(n))$ rééquilibrages chacuns coûtant $O(1)$ pour la suppression.

  En tout : $O(\log(n))$.

Récapitulatif pour les ABR équilibrés

- Les ABR équilibrés (rouges & noirs, AVL) permettent des opérations en $O(\log(n))$, où n est le nombre de nœuds.
- Permettent aussi, contrairement aux tables de hachage :
  - de trier les éléments en temps linéaire,
  - d'avoir accès en temps logarithmique au minimum, maximum.
  - d'avoir une complexité garantie.

Algorithmique des structures arborescentes
Algorithmique et programmation fonctionnelle

L2 Info et Math-info, 2019–20

Marc Zeitoun

27 février 2020

Organisation

- Aujourd'hui = avant-dernier cours en amphi.
- DS à venir.
- Groupes annoncés sous Moodle.

Plan

Parcours en profondeur préfixe

Écriture naturelle de façon récursive.

```
DFS(t):
si t est vide alors
  Retourner
sinon
  Traiter(t.racine)
  DFS(t.fils_gauche)
  DFS(t.fils_droit)
fin si
```

Parcours en profondeur préfixe

Écriture naturelle de façon récursive.

```
DFS(t):
si t est vide alors
  Retourner
sinon
  Traiter(t.racine)
  DFS(t.fils_gauche)
  DFS(t.fils_droit)
fin si
```

Comment adapter l'algorithme pour

- générer la **liste** des nœuds dans l'ordre préfixe,
- en complexité $O(n)$, où $n =$ nombre de nœuds ?

Parcours en profondeur préfixe

Écriture naturelle de façon récursive.

```
DFS(t):
si t est vide alors
  Retourner
sinon
  Traiter(t.racine)
  DFS(t.fils_gauche)
  DFS(t.fils_droit)
fin si
```

Comment adapter l'algorithme pour

- générer la **liste** des nœuds dans l'ordre préfixe,
- en complexité $O(n)$, où $n =$ nombre de nœuds ?

À éviter : Concaténations de listes $\rightsquigarrow O(n^2)$.

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $\boldsymbol{O}(1)$.

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $\boldsymbol{O}(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop

Push(1)

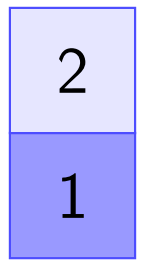
Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop



Push(1) Push(2)

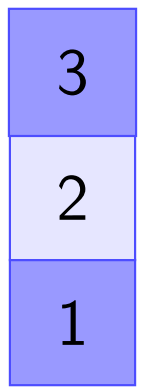
Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop



Push(1) Push(2) Push(3)

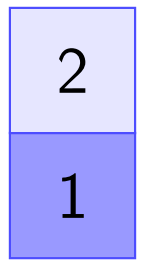
Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop



Push(1) Push(2) Push(3) Pop

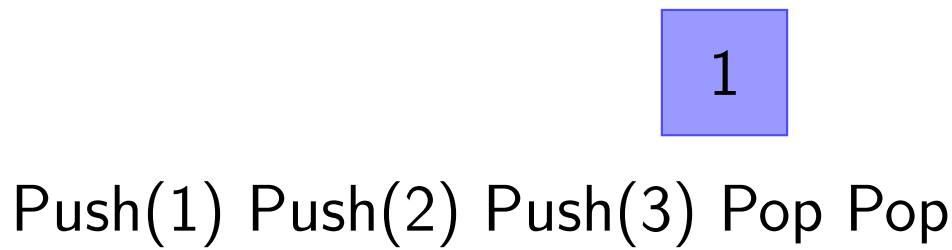
Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop



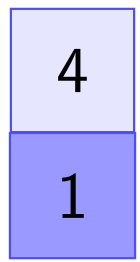
Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop



Push(1) Push(2) Push(3) Pop Pop Push(4)

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop

Push(1) Push(2) Push(3) Pop Pop Push(4) Pop

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop

Push(1) Push(2) Push(3) Pop Pop Push(4) Pop Pop

Parcours en profondeur préfixe

- 1re solution : utiliser des listes doublement chaînées.

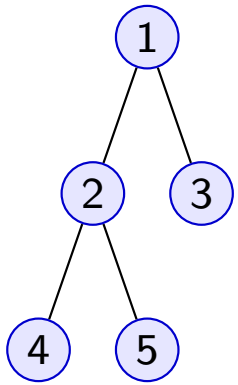
  $\rightsquigarrow$ concaténation de listes en $O(1)$.

- 2e solution : maintenir les sous-arbres non traités dans **une pile**.

Rappel sur les piles. 2 opérations : Push(x) et Pop

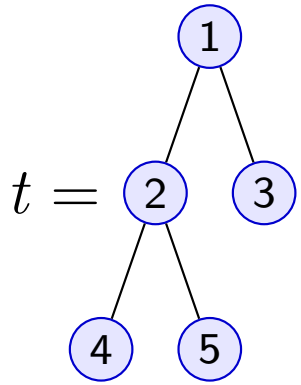
Pour le parcours, ce sont des sous-arbres qu'on mémorise

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
		t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;

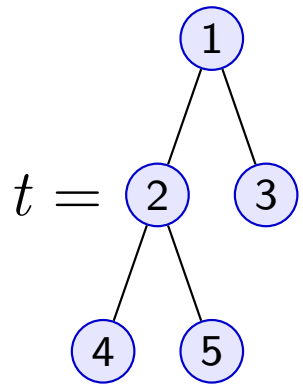
Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
		t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;



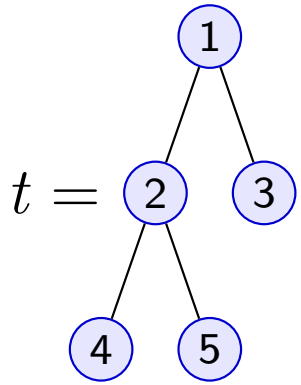
Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	1	Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;

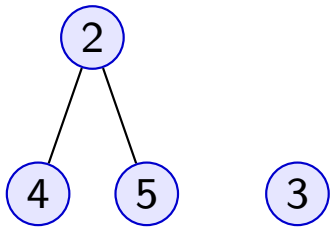


Parcours en profondeur préfixe

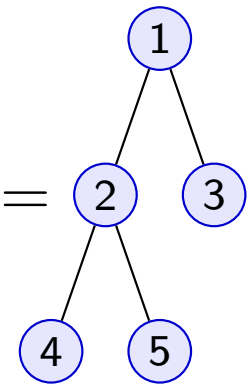
Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	1	
③		Push(t.right) ; Push(t.left) ;



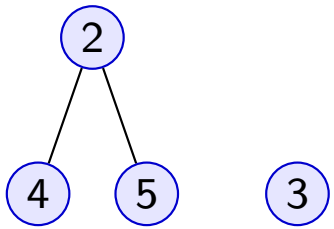
Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	1	
		Push(t.left) ;

$t =$

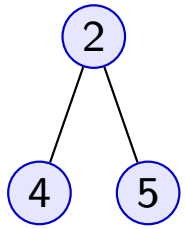


Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	1	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;

Parcours en profondeur préfixe

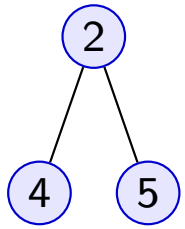
Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	1	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
3		

$t =$ 

Parcours en profondeur préfixe

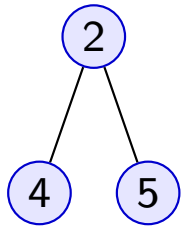
Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	2, **1**	Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
3		

$t =$



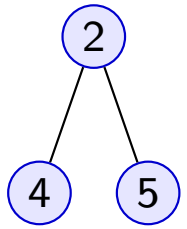
Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	2, **1**	
5 3		Push(t.right) ; Push(t.left) ;

$t =$ 

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	2, **1**	
4 5 3		Push(t.left) ;

$t =$ 

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	2, 1	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
4 5 3		

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	2, **1**	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
5 3		

$t =$ 4

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	4, **2**, **1**	Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
5 3		

$t =$ 4

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	4, **2**, **1**	
⑤ ③		Push(t.right) ; Push(t.left) ;

$t =$ ④

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	4, **2**, **1**	
5 3		Push(t.left) ;

$t =$ 4

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	4, **2**, **1**	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
(5) (3)		

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	4, **2**, **1**	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
3		

$t =$ 5

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	5, **4**, **2**, **1**	Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
3		

$t =$ 5

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	5, **4**, **2**, **1**	
③		Push(t.right) ; Push(t.left) ;

$t =$ ⑤

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	5, **4**, **2**, **1**	
③		Push(t.left) ;

$t =$ ⑤

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	5, **4**, **2**, **1**	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;
③		

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	5, **4**, **2**, **1**	t = Pop() ; Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;

$t =$ (3)

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	3, **5**, **4**, **2**, **1**	Accumuler(t.racine) ; Push(t.right) ; Push(t.left) ;

$t =$ 

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	3, **5**, **4**, **2**, **1**	Push(t.right) ; Push(t.left) ;

$t =$ 

Parcours en profondeur préfixe

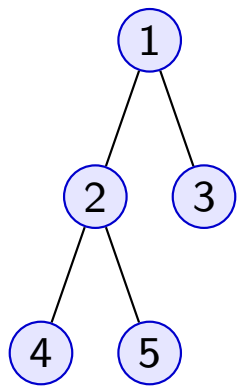
Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	3, **5**, **4**, **2**, **1**	
		Push(t.left) ;

$t =$ (3)

Parcours en profondeur préfixe

Pile d'arbres (haut de pile à gauche)	Accumulateur	Actions
	3, **5**, **4**, **2**, **1**	

Parcours en profondeur préfixe : 1, 2, 4, 5, 3



Parcours en profondeur : programmation

- On peut utiliser une liste pour simuler la pile.
- Les opérations Push et Pop s'implémentent en $O(1)$.

Parcours en profondeur : programmation

- On peut utiliser une liste pour simuler la pile.
- Les opérations Push et Pop s'implémentent en $O(1)$.
- Si la pile contient `t::fin` où `t=(Bin(x,left,right))` on retourne `DFS (left::right::fin) (x::acc)`

Parcours en profondeur : programmation

- On peut utiliser une liste pour simuler la pile.
- Les opérations Push et Pop s'implémentent en $O(1)$.
- Si la pile contient `t::fin` où `t=(Bin(x,left,right))` on retourne `DFS (left::right::fin) (x::acc)`
- La pile passe de `t::fin` à `left::right::fin`. On a :
  1. Dépilé `t`,
  2. puis empilé `right`,
  3. puis empilé `left`.

Autres parcours en profondeur : exercice

Adapter l'algorithme pour les parcours en profondeur **infixe**.

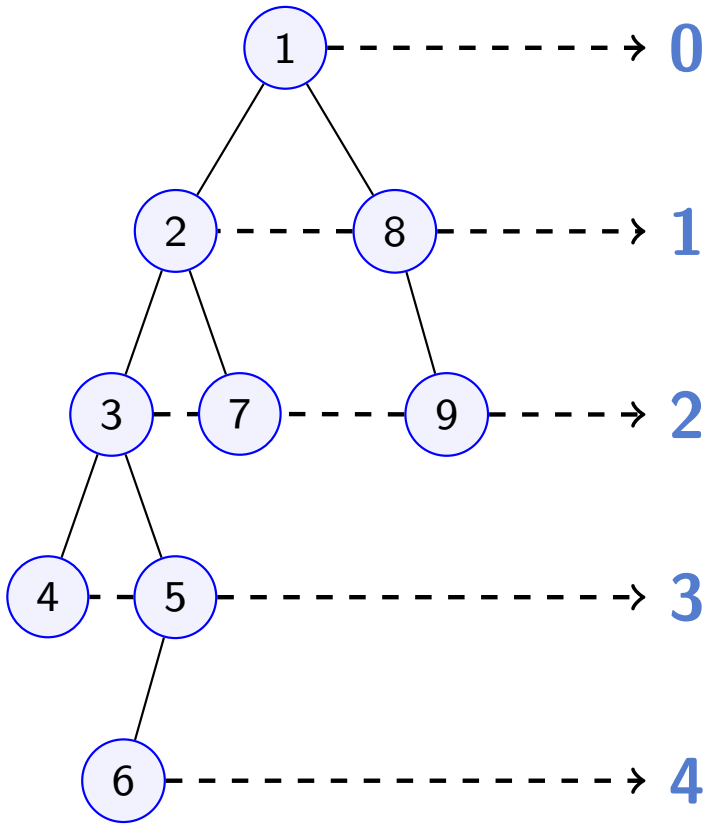
Parcours en largeur (BFS, Breadth-First Search)

Ce type de parcours visite les nœuds par **profondeurs croissantes**.

- On visite d'abord les nœuds à profondeur 0 (= la racine),
- Puis les nœuds à profondeur 1 (= les fils de la racine),
- Puis les nœuds à profondeur 2,
- etc.

Parcours en largeur : exemple

Visite des nœuds par **profondeurs croissantes**.



Parcours en largeur : 1 2 8 3 7 9 4 5 6

Parcours en largeur

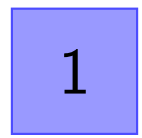
- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

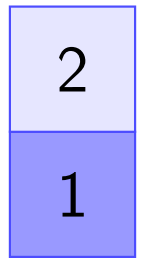


Enqueue(1)

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

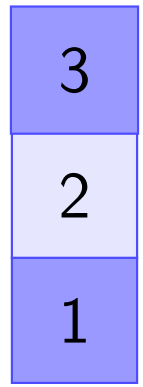


Enqueue(1) Enqueue(2)

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

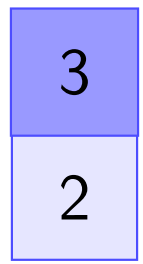


Enqueue(1) Enqueue(2) Enqueue(3)

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue



Enqueue(1) Enqueue(2) Enqueue(3) Dequeue

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

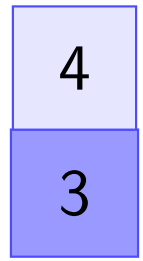
Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

Enqueue(1) Enqueue(2) Enqueue(3) Dequeue Dequeue

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue



Enqueue(1) Enqueue(2) Enqueue(3) Dequeue Dequeue Enqueue(4)

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

Enqueue(1) Enqueue(2) Enqueue(3) Dequeue Dequeue Enqueue(4) Dequeue

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

Enqueue(1) Enqueue(2) Enqueue(3) Dequeue Dequeue Enqueue(4)
Dequeue Dequeue

Parcours en largeur

- Même algorithme en remplaçant la **pile** par une **file**.

Rappel sur les files. 2 opérations : Enqueue(x) et Dequeue

Pour le parcours, ce sont des sous-arbres qu'on mémorise

Parcours en largeur

Parcours en largeur : programmation

- But : implémenter **Enqueue**, **Dequeue** en temps $O(1)$.
  - $\rightsquigarrow$ soit utiliser une liste doublement chaînée,
  - $\rightsquigarrow$ soit simuler la file par **2 piles**.

Parcours en largeur : programmation

- But : implémenter **Enqueue**, **Dequeue** en temps $O(1)$.
  $\rightsquigarrow$ soit utiliser une liste doublement chaînée,
  $\rightsquigarrow$ soit simuler la file par **2 piles**.
- Si la file contient `t::fin` où `t=(Bin(x,left,right))`
  on voudrait retourner `BFS (fin @ [left;right]) (x::acc)`

Parcours en largeur : programmation

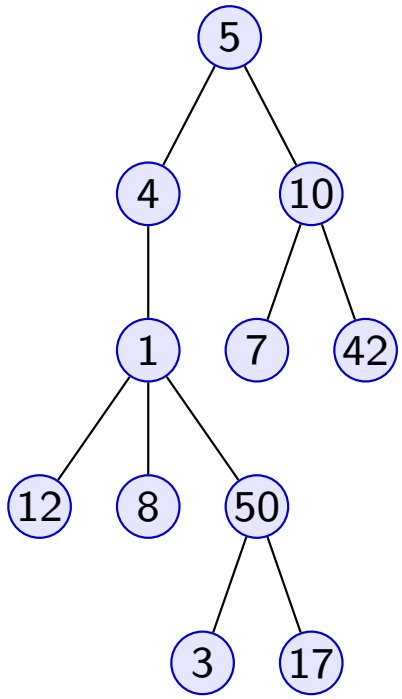
- But : implémenter **Enqueue**, **Dequeue** en temps $O(1)$.
  $\rightsquigarrow$ soit utiliser une liste doublement chaînée,
  $\rightsquigarrow$ soit simuler la file par **2 piles**.
- Si la file contient `t::fin` où `t=(Bin(x,left,right))`
  on voudrait retourner `BFS (fin @ [left;right]) (x::acc)`
- Non efficace à cause de la concaténation `@`.
- Implémentation efficace ? Simulation d'une file par **2 piles**.

Plan

Arbres planaires

Définition informelle : arbres dans lesquels

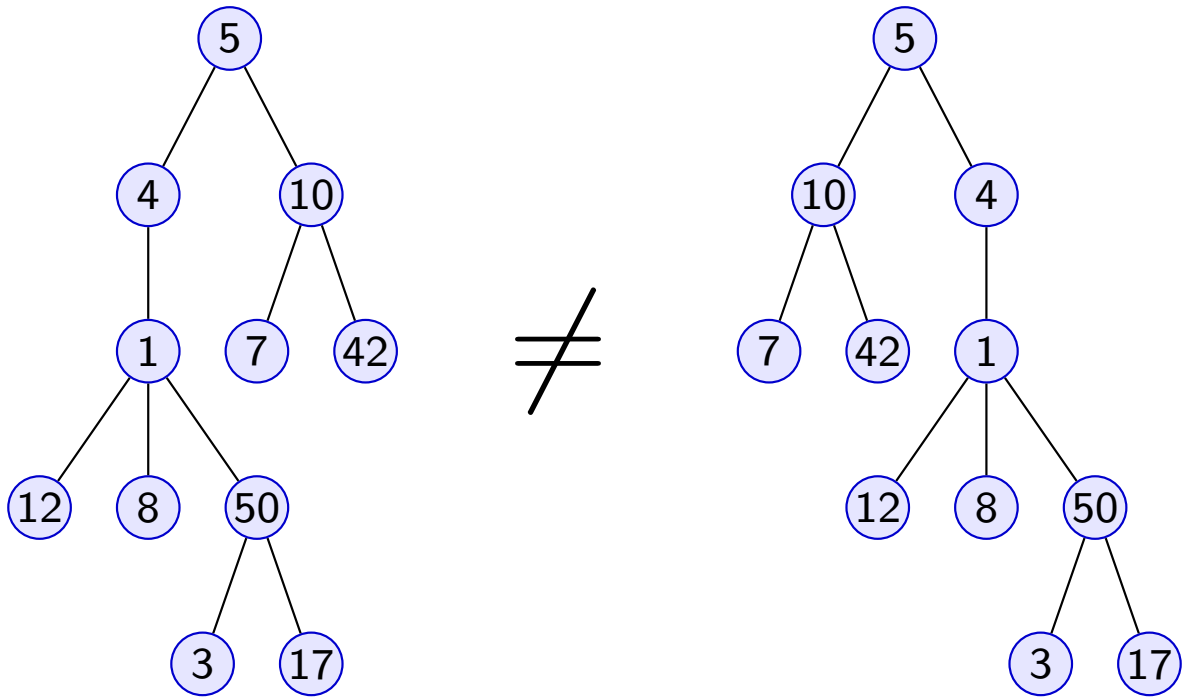
- l'arité n'est pas contrainte.
- l'ordre des fils à une importance.



Arbres planaires

Définition informelle : arbres dans lesquels

- l'arité n'est pas contrainte.
- l'ordre des fils à une importance.



Arbres planaires

Définition informelle : arbres dans lesquels

- l'arité n'est pas contrainte.
- l'ordre des fils à une importance.

Naturels (ex. arbres de décision en IA, arbres de récursion).

Arbres planaires

Définition informelle : arbres dans lesquels

- l'arité n'est pas contrainte.
- l'ordre des fils à une importance.

Naturels (ex. arbres de décision en IA, arbres de récursion).

Définition récursive des arbres planaires

- Constitué d'un nœud, et

Arbres planaires

Définition informelle : arbres dans lesquels

- l'arité n'est pas contrainte.
- l'ordre des fils à une importance.

Naturels (ex. arbres de décision en IA, arbres de récursion).

Définition récursive des arbres planaires

- Constitué d'un nœud, et
- d'une liste d'arbres planaires.

Arbres planaires

Définition informelle : arbres dans lesquels

- l'arité n'est pas contrainte.
- l'ordre des fils à une importance.

Naturels (ex. arbres de décision en IA, arbres de récursion).

Définition récursive des arbres planaires

- Constitué d'un nœud, et
- d'une liste d'arbres planaires.

Note. Cette définition exclut l'arbre vide.

Plan

Énumération (de squelettes) d'arbres

Combien y a-t-il de squelettes

- d'arbres binaires à n nœuds ?
- d'arbres binaires complets à n nœuds ?
- d'arbres planaires à n nœuds ?

Énumération (de squelettes) d'arbres

Combien y a-t-il de squelettes

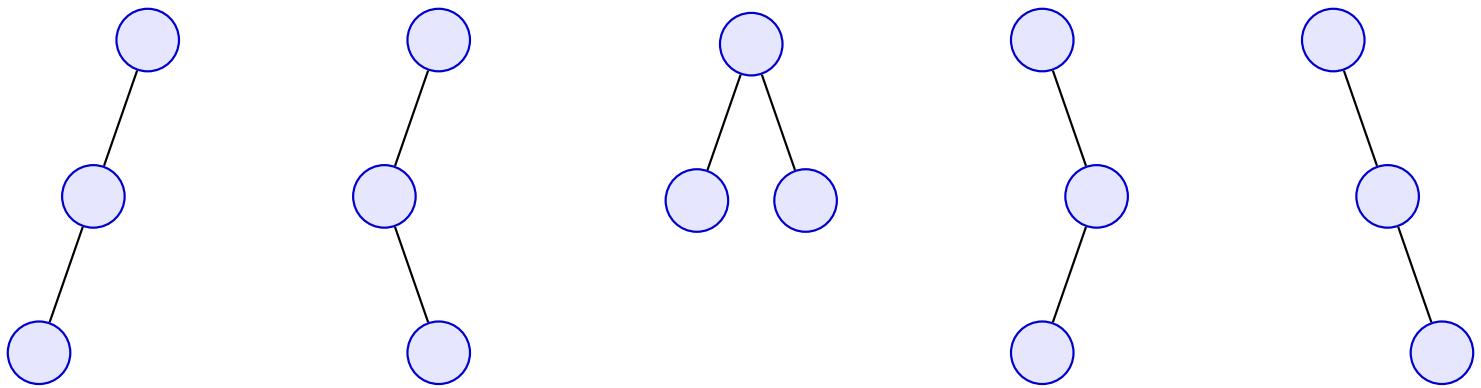
- d'arbres binaires à n nœuds ?
- d'arbres binaires complets à n nœuds ?
- d'arbres planaires à n nœuds ?

Intérêt de ces questions

- représentation différentes de mêmes objets,
- génération aléatoire,
- analyse d'algorithmes **en moyenne**.

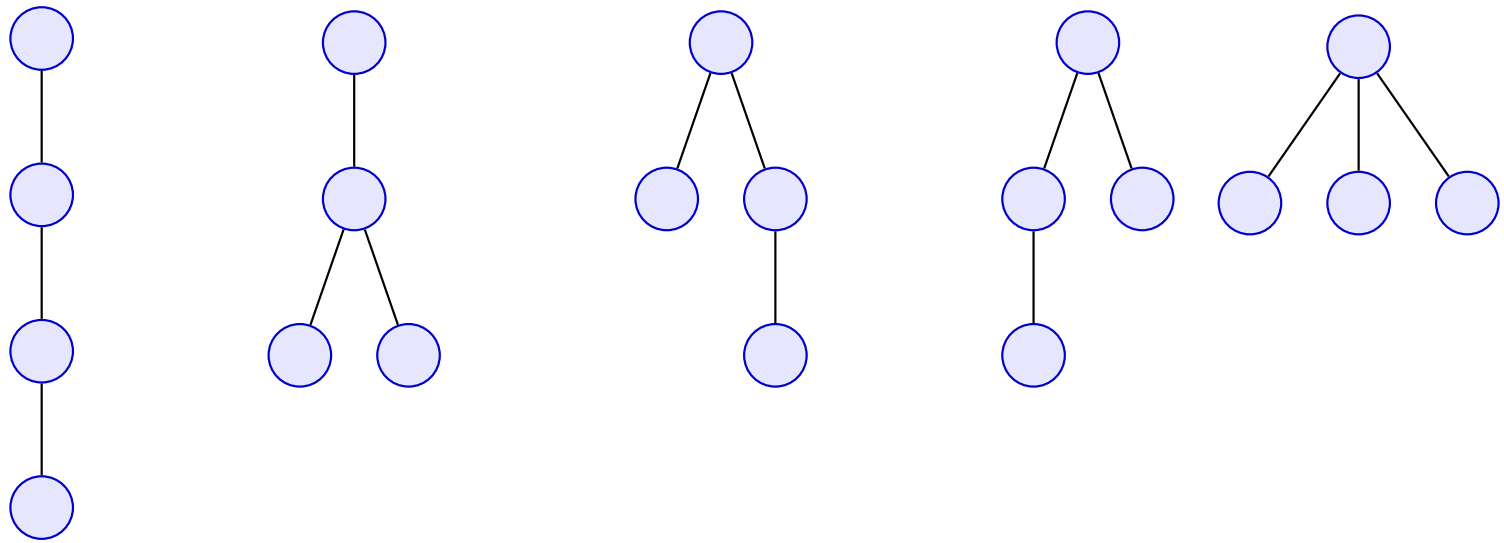
Énumération d'arbres binaires

Combien y a -t-il de squelettes d'arbres **binaires à 3 nœuds** ?



Énumération d'arbres planaires

Combien y a -t-il de squelettes d'arbres **planaires à 4 nœuds** ?



Comptage

Il y a autant

- de squelettes d'arbres binaires à n nœuds.
- de squelettes d'arbres binaires complets à $2n+1$ nœuds.
- de squelettes d'arbres planaires à $n+1$ nœuds.
- de mots de Dyck de longueur $2n$.
- de chemins de Dyck de longueur $2n$.

Comptage

Il y a autant

- de squelettes d'arbres binaires à n nœuds.
- de squelettes d'arbres binaires complets à $2n+1$ nœuds.
- de squelettes d'arbres planaires à $n+1$ nœuds.
- de mots de Dyck de longueur $2n$.
- de chemins de Dyck de longueur $2n$.

Leur nombre est

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Nombres de Catalan [Lien 1,2] : 1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Comptage

Il y a autant

- de squelettes d'arbres binaires à n nœuds.
- de squelettes d'arbres binaires complets à $2n+1$ nœuds.
- de squelettes d'arbres planaires à $n+1$ nœuds.
- de mots de Dyck de longueur $2n$.
- de chemins de Dyck de longueur $2n$.

Leur nombre est

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

Nombres de Catalan [Lien 1,2] : 1, 1, 2, 5, 14, 42, 132, 429, 1430, ...

Il y a des **bijections simples** entre ces ensembles.

Mots de Dyck

Un **mot de Dyck** est une suite de symboles $\uparrow, \downarrow$ tq :

1. Il y a autant de $\uparrow$ que de $\downarrow$.
2. Quand on lit la suite de gauche à droite, il y a toujours au moins autant de $\uparrow$ que de $\downarrow$.

Exemple

- $\uparrow\uparrow\downarrow\downarrow$
- $\uparrow\downarrow\uparrow\downarrow$

Mots de Dyck

Un **mot de Dyck** est une suite de symboles $\uparrow, \downarrow$ tq :

1. Il y a autant de $\uparrow$ que de $\downarrow$.
2. Quand on lit la suite de gauche à droite, il y a toujours au moins autant de $\uparrow$ que de $\downarrow$.

Exemple

- $\uparrow\uparrow\downarrow\downarrow$ (())
- $\uparrow\downarrow\uparrow\downarrow$ ()()

Un mot de Dyck est aussi appelé mot bien parenthésé.

Mots de Dyck

Un **mot de Dyck** est une suite de symboles $\uparrow, \downarrow$ tq :

1. Il y a autant de $\uparrow$ que de $\downarrow$.
2. Quand on lit la suite de gauche à droite, il y a toujours au moins autant de $\uparrow$ que de $\downarrow$.

Exemple

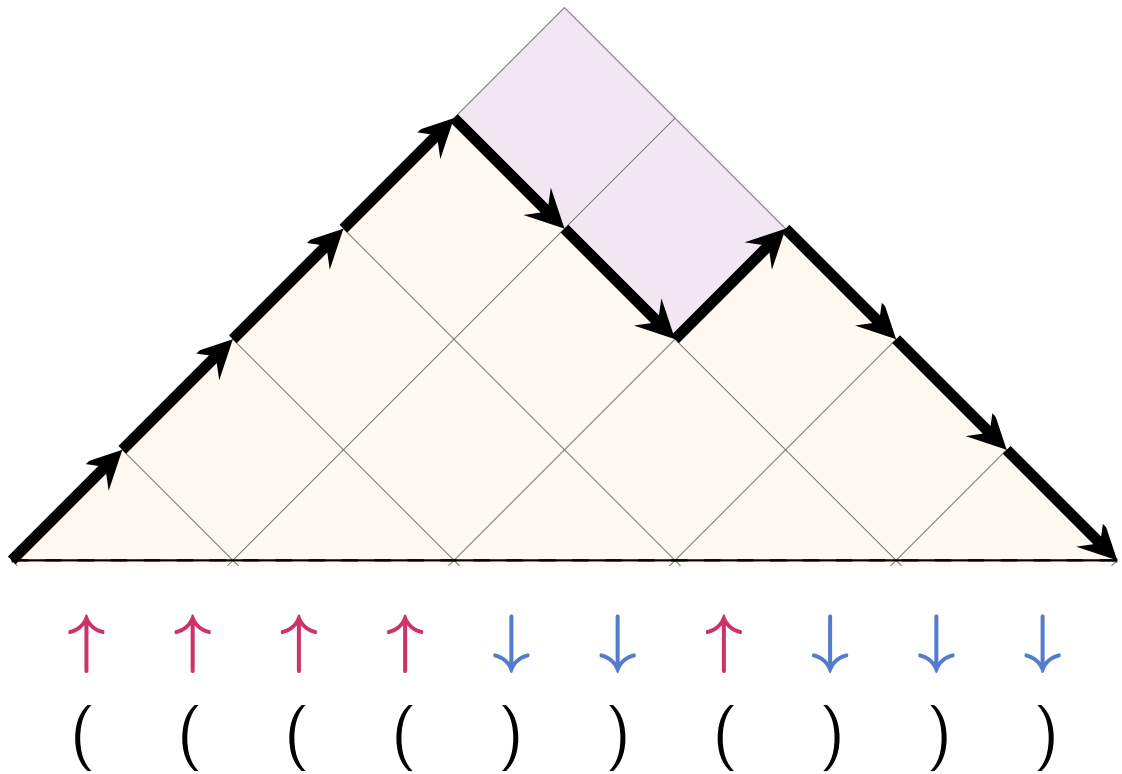
- $\uparrow\uparrow\downarrow\downarrow$ $\quad(())$
- $\uparrow\downarrow\uparrow\downarrow$ $\quad()()$

Un mot de Dyck est aussi appelé mot bien parenthésé.

1. Autant d'ouvrantes que de fermantes.
2. Jamais plus de fermantes que d'ouvrantes quand on lit de gauche à droite.

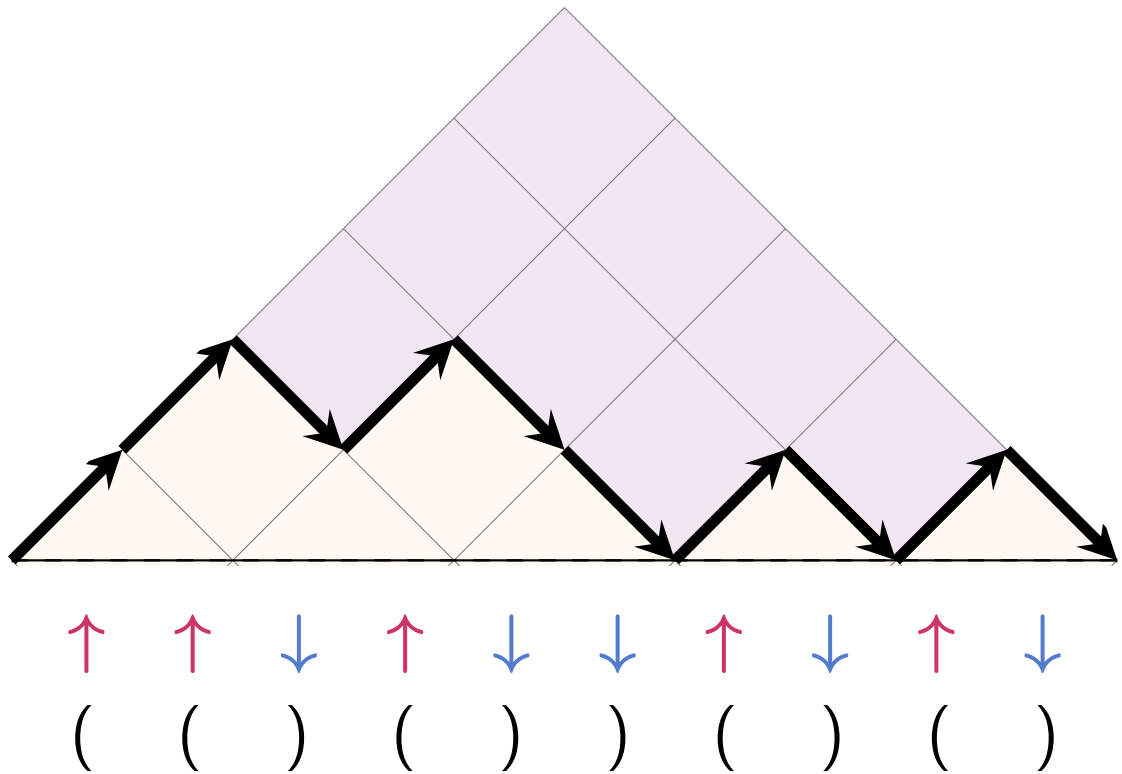
Chemins de Dyck

Simplement une représentation graphique des mots de Dyck.



Chemins de Dyck

Simplement une représentation graphique des mots de Dyck.

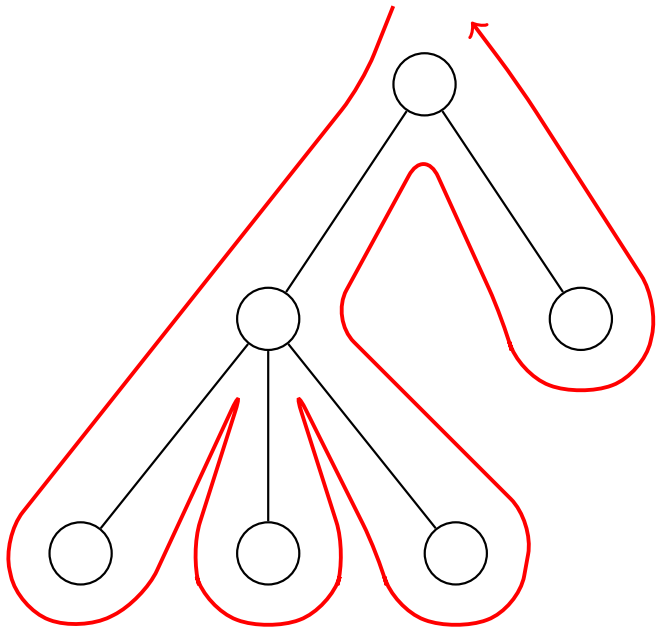


Chemins de Dyck

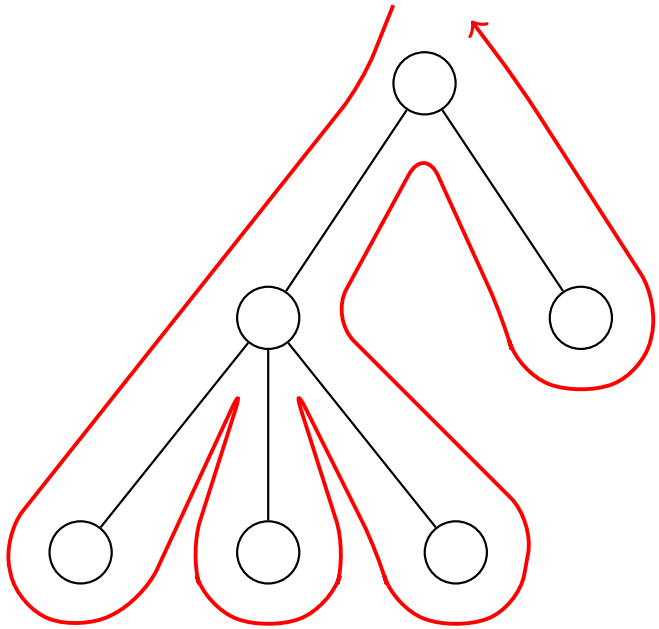
Ces chemins

- Partent de l'origine (0,0),
- 2 types de pas : haut ou bas.
- Restent au dessus de l'altitude 0.
- Terminent à l'altitude 0.

Bijection arbres planaires $\leftrightarrow$ chemins de Dyck

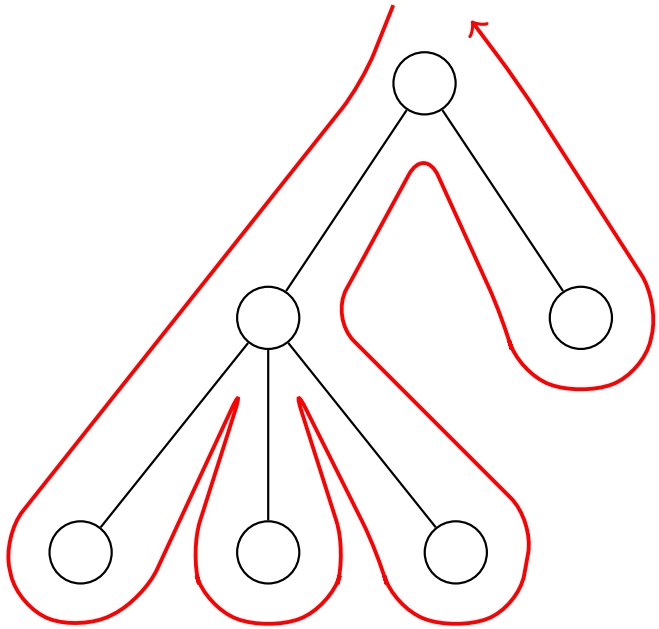


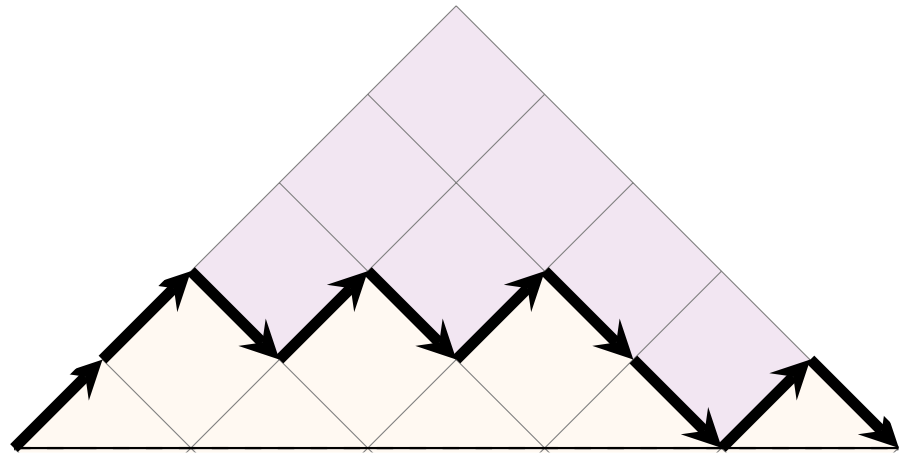
Bijection arbres planaires $\leftrightarrow$ chemins de Dyck



- Parcours en profondeur de l'arbre.
- On produit (lorsqu'on descend et) lorsqu'on monte.

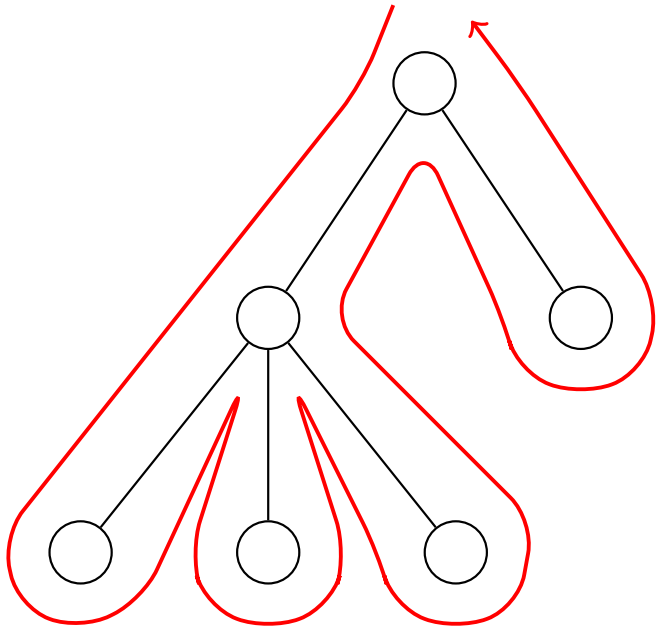
Bijection arbres planaires $\leftrightarrow$ chemins de Dyck

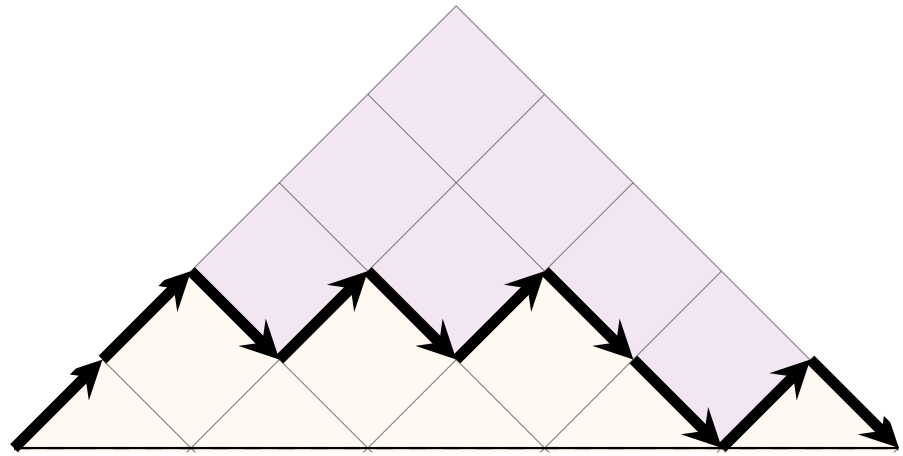




- Parcours en profondeur de l'arbre.
- On produit (lorsqu'on descend et) lorsqu'on monte.

Bijection arbres planaires $\leftrightarrow$ chemins de Dyck





C'est une **bijection**

- Chaque arbre donne un chemin différent,
- Chaque chemin est obtenu depuis un arbre.

Arbres planaires représentés par arbres binaires

Représentation simple d'un arbre planaire : associer à chaque nœud

- son fils gauche,
- son frère droit.

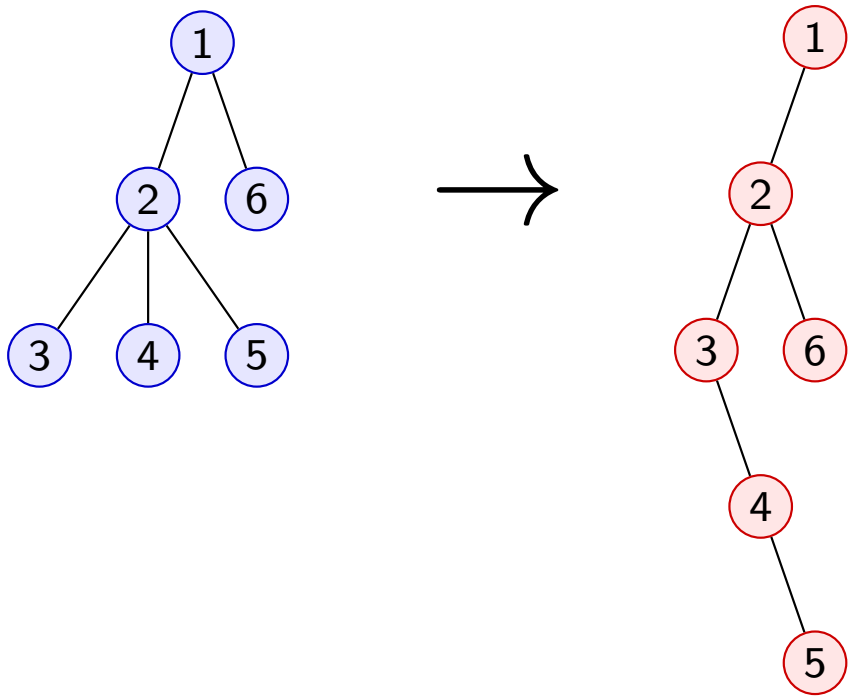
Par exemple, en C, on pourrait définir

```
struct planar_tree
{
  int key;
  struct planar_tree *left_child, *next_brother;
};
```

Ce codage donne l'idée d'une bijection.

Arbres planaires représentés par arbres binaires

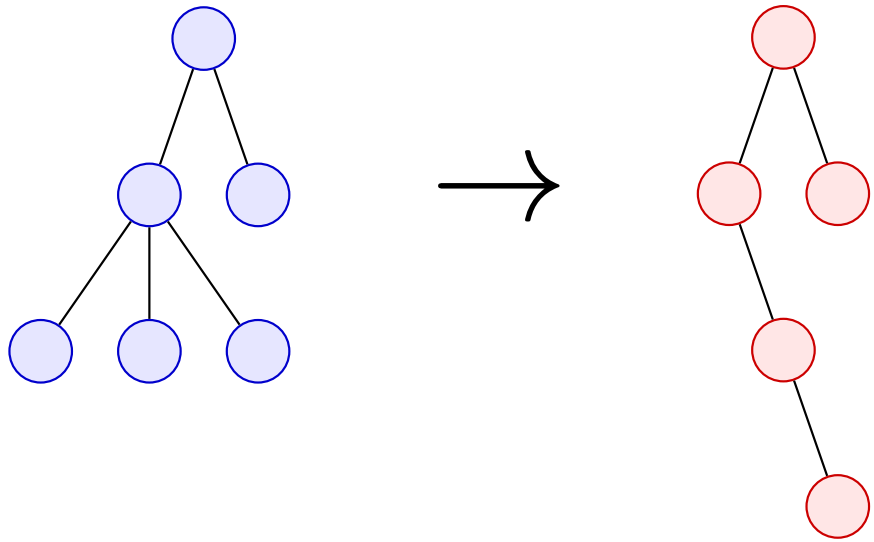
On donne comme fils d'un nœud dans l'arbre binaire son fils gauche et son frère droit dans l'arbre planaire.



La racine de l'arbre binaire obtenu n'a jamais de fils droit.

Arbres planaires représentés par arbres binaires

Pour obtenir une bijection, entre arbres planaires à $n+1$ nœuds et arbres binaires à n nœuds, on supprime la racine de l'arbre obtenu.



Nombre de chemins de Dyck de taille $2n$

- Il y a chemins de taille $2n$ allant de $(0, 0)$ à $(2n, 0)$.

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.
- On doit retrancher les chemins descendant sous l'altitude 0.

  $\Rightarrow$ Combien de chemins descendent sous l'altitude 0 ?

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.
- On doit retrancher les chemins descendant sous l'altitude 0.

  $\Rightarrow$ Combien de chemins descendent sous l'altitude 0 ?

Autant que de chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.
- On doit retrancher les chemins descendant sous l'altitude 0.

  $\Rightarrow$ Combien de chemins descendent sous l'altitude 0 ?

Autant que de chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

En effet, il y a une **bijection** entre les ensembles

- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$ passant au moins une fois sous l'altitude 0.
- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.
- On doit retrancher les chemins descendant sous l'altitude 0.

  $\Rightarrow$ Combien de chemins descendent sous l'altitude 0 ?

Autant que de chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

En effet, il y a une **bijection** entre les ensembles

- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$ passant au moins une fois sous l'altitude 0.
- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

- Il y a tels chemins (choix des $n+1$ descentes).

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.
- On doit retrancher les chemins descendant sous l'altitude 0.

  $\Rightarrow$ Combien de chemins descendent sous l'altitude 0 ?

Autant que de chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

En effet, il y a une **bijection** entre les ensembles

- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$ passant au moins une fois sous l'altitude 0.
- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

- Il y a $\binom{2n}{n+1}$ tels chemins (choix des $n+1$ descentes).

Nombre de chemins de Dyck de taille $2n$

- Il y a $\binom{2n}{n}$ chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$.
- On doit retrancher les chemins descendant sous l'altitude 0.

  $\Rightarrow$ Combien de chemins descendent sous l'altitude 0 ?

Autant que de chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

En effet, il y a une **bijection** entre les ensembles

- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$ passant au moins une fois sous l'altitude 0.
- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

- Il y a $\binom{2n}{n+1}$ tels chemins (choix des $n+1$ descentes).
- Le nombre de chemins de Dyck de taille $2n$ est donc

$$\binom{2n}{n} - \binom{2n}{n+1} = \frac{1}{n+1}\binom{2n}{n}.$$

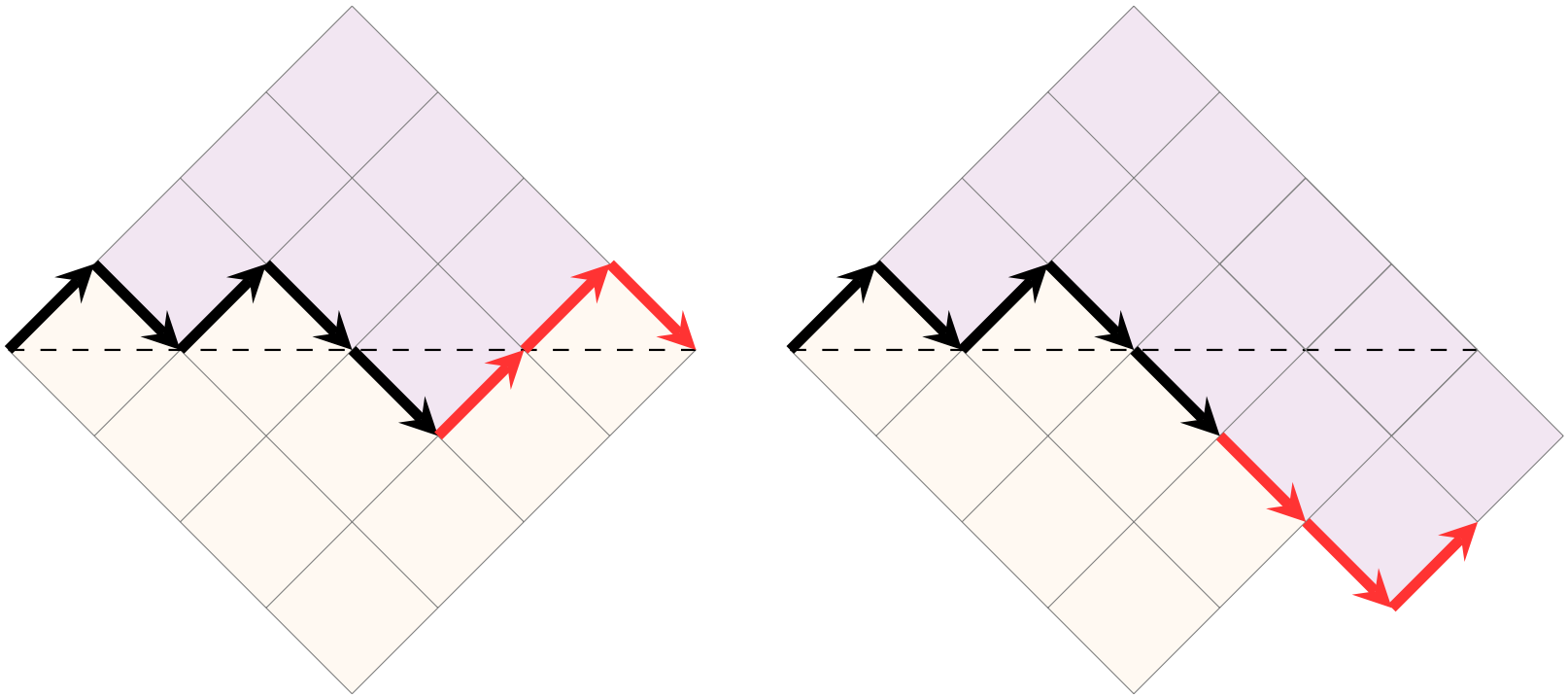
Chemins descendant sous l'altitude 0

Bijection entre les ensembles

- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$ passant au moins une fois sous l'altitude 0.
- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

Bijection : inversion des pas à partir du premier point d'altitude -1.

Exemple 1



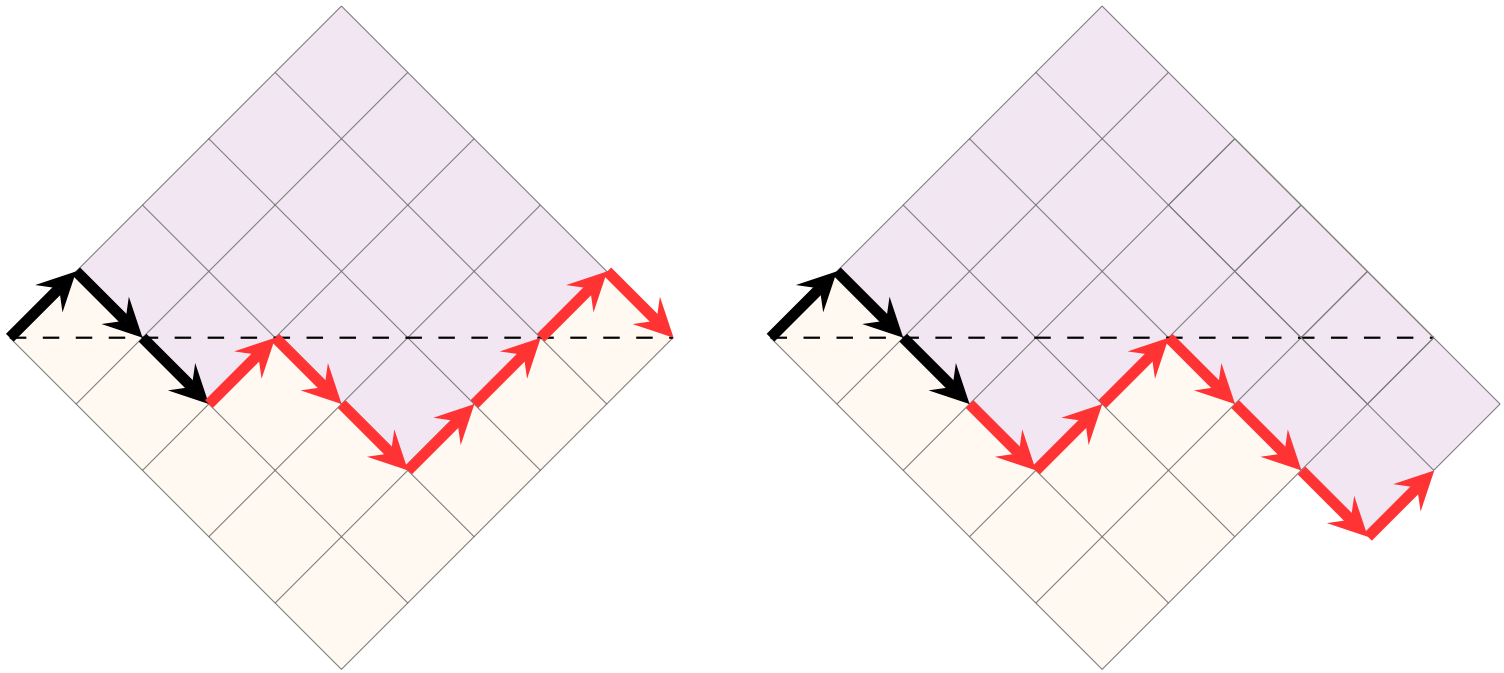
Chemins descendant sous l'altitude 0

Bijection entre les ensembles

- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,0)$ passant au moins une fois sous l'altitude 0.
- des chemins de taille $2n$ allant de $(0,0)$ à $(2n,-2)$.

Bijection : inversion des pas à partir du premier point d'altitude -1.

Exemple 2



Algorithmique des structures arborescentes
Algorithmique et programmation fonctionnelle

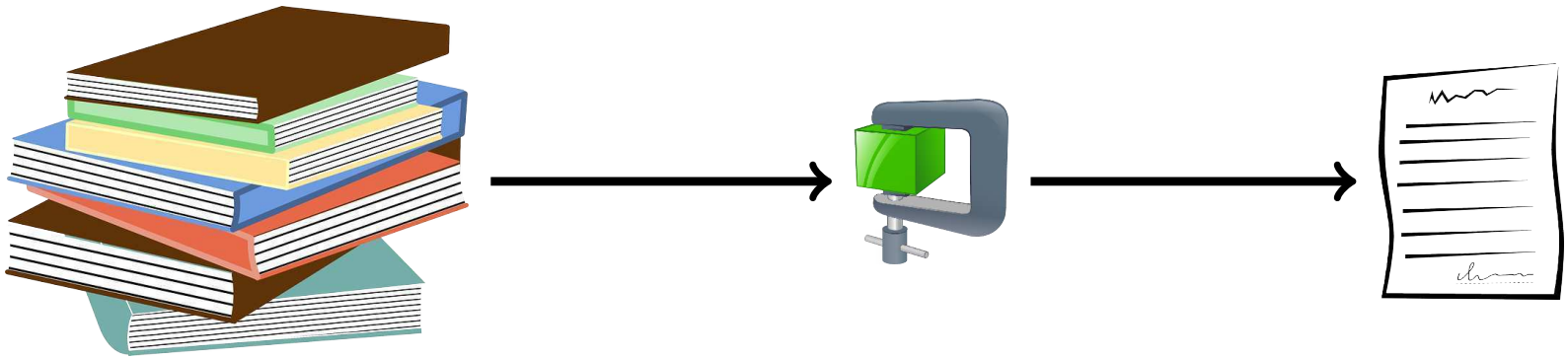
L2 Info et Math-info, 2019–20

Marc Zeitoun

2 avril 2020

Plan

Compression sans perte

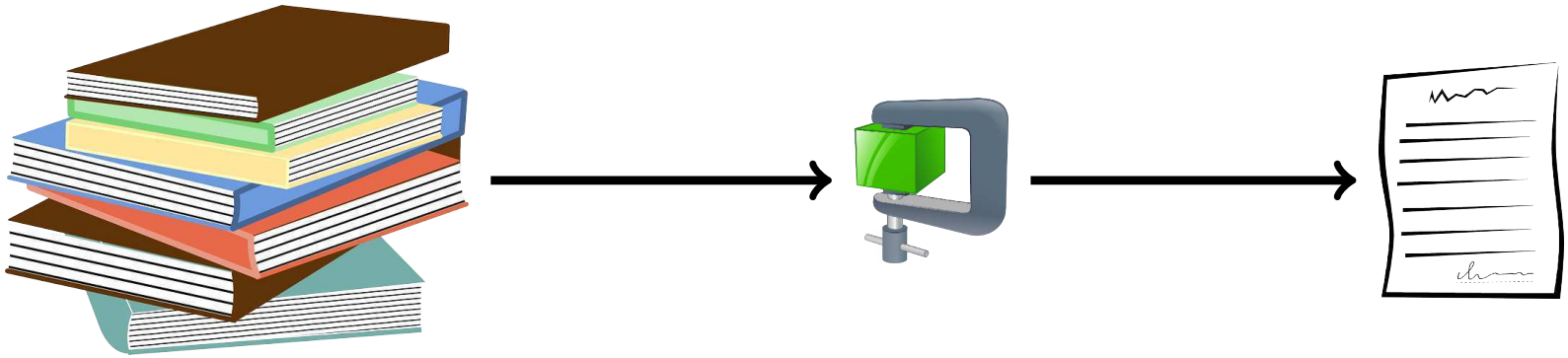


1Tb **25Kb**

Objectifs :

- Réduire la taille des fichiers.

Compression sans perte

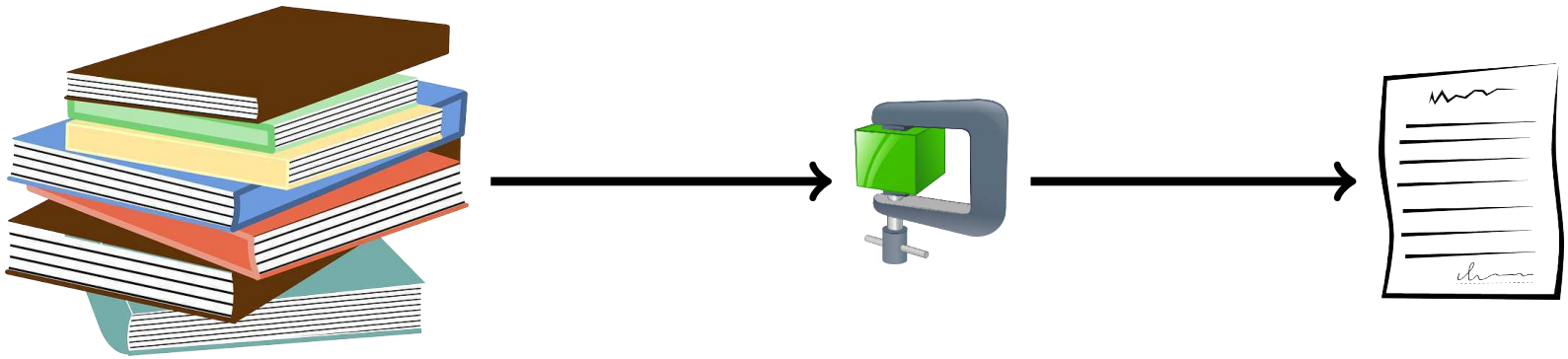


1Tb **25Kb**

Objectifs :

- Réduire la taille des fichiers.
- **Sans perte** : on veut pouvoir reconstruire le fichier **original**.

Compression sans perte



1Tb **25Kb**

Objectifs :

- Réduire la taille des fichiers.
- **Sans perte** : on veut pouvoir reconstruire le fichier **original**.
- Aujourd'hui : 2 algorithmes : LZ78 et Huffman.
  Implémentation utilisant des arbres binaires.

Codages de caractères

- Un fichier sur disque est une suite de caractères.

Codages de caractères

- Un fichier sur disque est une suite de caractères.
- Un codage associe à chaque caractère une **représentation** par un ou plusieurs **octets**.

Codages de caractères

- Un fichier sur disque est une suite de caractères.
- Un codage associe à chaque caractère une **représentation** par un ou plusieurs **octets**.
- Il y a plusieurs codages existants. Par exemple,
  - ASCII : représente 128 caractères chacun **sur un octet**.

Codages de caractères

- Un fichier sur disque est une suite de caractères.
- Un codage associe à chaque caractère une **représentation** par un ou plusieurs **octets**.
- Il y a plusieurs codages existants. Par exemple,
  - ASCII : représente 128 caractères chacun **sur un octet**.
  - ISO 8859-1 (latin1) l'étend à 191 caractères **sur un octet**.

Codages de caractères

- Un fichier sur disque est une suite de caractères.
- Un codage associe à chaque caractère une **représentation** par un ou plusieurs **octets**.
- Il y a plusieurs codages existants. Par exemple,
  - ASCII : représente 128 caractères chacun **sur un octet**.
  - ISO 8859-1 (latin1) l'étend à 191 caractères **sur un octet**.
  - UTF-8 pour les caractères du standard Unicode : 1 à 4 octets.
    $\Longrightarrow$ Plus difficile de décoder un fichier UTF-8.

Plan

Compression de Huffman : principe

- Idée : coder les caractères sur un **nombre variable de bits**, éventuellement moins que 8.
- Caractères **fréquents** représentés par des codes **courts**.
- Caractères **peu fréquents** représentés par des codes **longs**.
- **Difficulté** trouver un codage permettant de décoder.
- Par exemple, si on code le caractère a par 0 et b par 00, on ne pourra pas décoder la suite 000 : elle peut représenter ab ou ba.

Compression de Huffman : étapes

1. Calcul des fréquences de caractères dans le fichier à compresser.

Compression de Huffman : étapes

1. Calcul des fréquences de caractères dans le fichier à compresser.
2. Calcul du codage de chaque caractère, avec 2 contraintes :

Compression de Huffman : étapes

1. Calcul des fréquences de caractères dans le fichier à compresser.
2. Calcul du codage de chaque caractère, avec 2 contraintes :
   - caractères les plus fréquents représentés par des codes **courts**,

Compression de Huffman : étapes

1. Calcul des fréquences de caractères dans le fichier à compresser.
2. Calcul du codage de chaque caractère, avec 2 contraintes :
   - caractères les plus fréquents représentés par des codes **courts**,
   - possibilité de décoder.

Compression de Huffman : étapes

1. Calcul des fréquences de caractères dans le fichier à compresser.
2. Calcul du codage de chaque caractère, avec 2 contraintes :
   - caractères les plus fréquents représentés par des codes **courts**,
   - possibilité de décoder.
3. Dans le fichier compressé, écrire le codage utilisé.

Compression de Huffman : étapes

1. Calcul des fréquences de caractères dans le fichier à compresser.
2. Calcul du codage de chaque caractère, avec 2 contraintes :
   - caractères les plus fréquents représentés par des codes **courts**,
   - possibilité de décoder.
3. Dans le fichier compressé, écrire le codage utilisé.
4. Enfin, relire le fichier original, et pour chacun de ses caractères, écrire son code dans le fichier compressé.

   Difficulté : la plus petite quantité qu'on peut écrire : **1 octet**.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.
  Par exemple, **01** est préfixe de **01**001.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.
  Par exemple, **01** est préfixe de **01**001.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.
  Par exemple, **01** est préfixe de **01**001.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\quad a \quad b \quad c \quad d \quad r}{C = \{000, 010, 011, 10, 11\}}$.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\begin{matrix} a & b & c & d & r \end{matrix}}{C = \{000, 010, 011, 10, 11\}}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\begin{matrix} a & b & c & d & r \end{matrix}}{C = \{000, 010, 011, 10, 11\}}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot a

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\quad}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot ab

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abr

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110001100010000001011000 code le mot abra

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\quad a \quad b \quad c \quad d \quad r}{C = \{000, 010, 011, 10, 11\}}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abrac

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\begin{matrix} a & b & c & d & r \end{matrix}}{C = \{000, 010, 011, 10, 11\}}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abraca

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\quad}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abracad

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\quad}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abracada

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abracadab

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abracadabr

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 000010110000110001000001011000 code le mot abracadabra

Codes préfixes

- Appelons **mot** une suite de **0** et de **1**. Par exemple **01001**.
- Un mot x est **préfixe** d'un mot y si y commence par x.

  Par exemple, **01** est préfixe de **01**001.
- Un ensemble fini de mots C est un **code préfixe** si aucun mot de C n'est préfixe d'un autre mot de C.
- Exemple : $\underset{\substack{\\ a}}{C} = \{\underset{a}{000}, \underset{b}{010}, \underset{c}{011}, \underset{d}{10}, \underset{r}{11}\}$.
- Un mot ne peut pas se décomposer de 2 façons différentes.
- On décode en lisant la suite de 0 et de 1 **de gauche à droite**.
- **Exemple** : la suite 0000101100001100010000001011000 code le mot abracadabra

  30 bits au lieu de $11 \times 8 = 88$ bits. **Peut-on faire mieux** ?

Codes préfixes et arbres binaires

- Un **code préfixe** peut être représenté par un **arbre binaire**.
- Chaque feuille correspond à un mot du code : on part de la racine jusqu'à la feuille, en écrivant **0** à chaque fois qu'on descend à gauche et **1** à chaque fois qu'on descend à droite.
- Chaque feuille correspond à un caractère du fichier à coder.
- Exemple : $C = \{000, 010, 011, 10, 11\}$.

Codage de Huffman : exemple

On veut coder le texte MISSISSIPPI MAP

Difficultés du codage de Huffman

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.
- $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$: fréquences correspondantes.

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.
- $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$: fréquences correspondantes.
- $\ell_1, \ell_2, \ldots, \ell_n$ = longueurs des codes de $s_1, \ldots, s_n$.

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.
- $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$: fréquences correspondantes.
- $\ell_1, \ell_2, \ldots, \ell_n$ = longueurs des codes de $s_1, \ldots, s_n$.
- Espérance de la longueur d'un code :

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.
- $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$: fréquences correspondantes.
- $\ell_1, \ell_2, \ldots, \ell_n$ = longueurs des codes de $s_1, \ldots, s_n$.
- Espérance de la longueur d'un code :

$$L = \sum_{k=1}^{n} f_i \ell_i$$

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.
- $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$: fréquences correspondantes.
- $\ell_1, \ell_2, \ldots, \ell_n$ = longueurs des codes de $s_1, \ldots, s_n$.
- Espérance de la longueur d'un code :

$$L = \sum_{k=1}^{n} f_i \ell_i$$

Quelle est la longueur du texte produit ?

Codes optimaux

Le codage de Huffman est **optimal** au sens suivant :

- Symboles originaux : $s_1, \ldots, s_n$.
- $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$: fréquences correspondantes.
- $\ell_1, \ell_2, \ldots, \ell_n$ = longueurs des codes de $s_1, \ldots, s_n$.
- Espérance de la longueur d'un code :

$$L = \sum_{k=1}^{n} f_i \ell_i$$

Quelle est la longueur du texte produit ?

Le codage de Huffman produit **l'espérance minimale**.

Optimalité du codage de Huffman

Plan

Codage LZ78

Compression par dictionnaire :

- On représente par un code les sous-chaînes du texte.

Codage LZ78

Compression par dictionnaire :

- On représente par un code les sous-chaînes du texte.
- Historique : LZ77, LZ78, LZW.

Codage LZ78

Compression par dictionnaire :

- On représente par un code les sous-chaînes du texte.
- Historique : LZ77, LZ78, LZW.
- Développé par Lempel et Ziv.

Codage LZ78

Compression par dictionnaire :

- On représente par un code les sous-chaînes du texte.
- Historique : LZ77, LZ78, LZW.
- Développé par Lempel et Ziv.
- Amélioré (et breveté) par Welsh.

Codage LZ78

Compression par dictionnaire :

- On représente par un code les sous-chaînes du texte.
- Historique : LZ77, LZ78, LZW.
- Développé par Lempel et Ziv.
- Amélioré (et breveté) par Welsh.
- gzip, zip, GIF, TIFF,...

Codage LZ78 : principe et exemple

00001011000011000100000101100 0

Taux de compression minimal et maximal

- Une chaîne qui se compresse bien ?

Taux de compression minimal et maximal

- Une chaîne qui se compresse bien ?

- Taux de compression ?

Taux de compression minimal et maximal

- Une chaîne qui se compresse bien ?
- Taux de compression ?
- Une chaîne qui se compresse mal ?

Taux de compression minimal et maximal

- Une chaîne qui se compresse bien ?
- Taux de compression ?
- Une chaîne qui se compresse mal ?
- Taux de compression ?