Algorithmique
&
Complexité

Merci à Maxime CROCHEMORE

Notion d'efficacité

- Plusieurs chemins pour une même solution
  - Comment choisir ?
  - Problèmes de **grande taille**

- Plusieurs solutions
  - Implémenter et tester les différentes solutions
  - **Prédire** le temps/coût nécessaire

- Important : la notion de taille du problème

Pourquoi se soucier de l'efficacité ?

- Mémoire : possibilité de calculer au pas

- Temps :
  - 1 min, 1 jour, 1 an de calcul ?
  - 2x plus de donnée = 4x plus de temps ? 16x ?

- Consommation électrique
  - Embarqué
  - ……

Exemple 1 : calculer x^n

- Méthode 1 :
  - calculer le nombre d'affectations
  - calculer le nombre de multiplications

```
pow1(in réel x, in entierpos n) : réel
    entierpos r
    r ← 1
    pour i=1 à n faire
        r ← r*x
    fin pour
    retourner r
```

n *

n+1 ←

...

Exemple 1 : calculer x^n

- Méthode 2

```
pow2(in réel x, in entierpos n) : réel
    entierpos r
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow2(x,n/2)
        r ← r*r
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

~ $2 \log_2(n+1)$ *

~ $3 \log_2(n+1) + 1$ ←

~ $\log_2(n)$ appels à pow2

...

Analyse de la complexité

- Définition – complexité d'un algorithme
  - mesure du nombre d'**opérations fondamentales** qu'il effectue sur un jeu de données.
  - exprimée comme une fonction de la taille du jeu de données.

- **Ordre de grandeur**
  - Pas la complexité exacte
    - Comportement **asymptotique**

Comportement asymptotique : rappel limites

$$\lim_{n\to\infty} n^2+n+1 = \lim_{n\to\infty} n^2$$

$$\lim_{n\to\infty} \ln(n)+n = \lim_{n\to\infty} n$$

$$\lim_{n\to\infty} \sqrt{n}+n = \lim_{n\to\infty} n$$

$$\lim_{n\to\infty} x^n+n = \lim_{n\to\infty} x^n$$

Analyse de la complexité

- Définition – complexité d'un algorithme
  - mesure du nombre d'**opérations fondamentales** qu'il effectue sur un jeu de données.
  - exprimée comme une fonction de la taille du jeu de données.

- **Ordre de grandeur**
  - Pas la complexité exacte

    Comportement **asymptotique**
  - En général, borne supérieure

Exemple 1 : calculer x^n

- Méthode 2

```
pow2(in réel x, in entierpos n) : réel
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow2(x,n/2)
        r ← r*r
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

~ $2 \log_2(n+1)$ *

~ $3 \log_2(n+1) + 1$ ←

~ $\log_2(n)$ appels à pow2

...

Analyse de la complexité

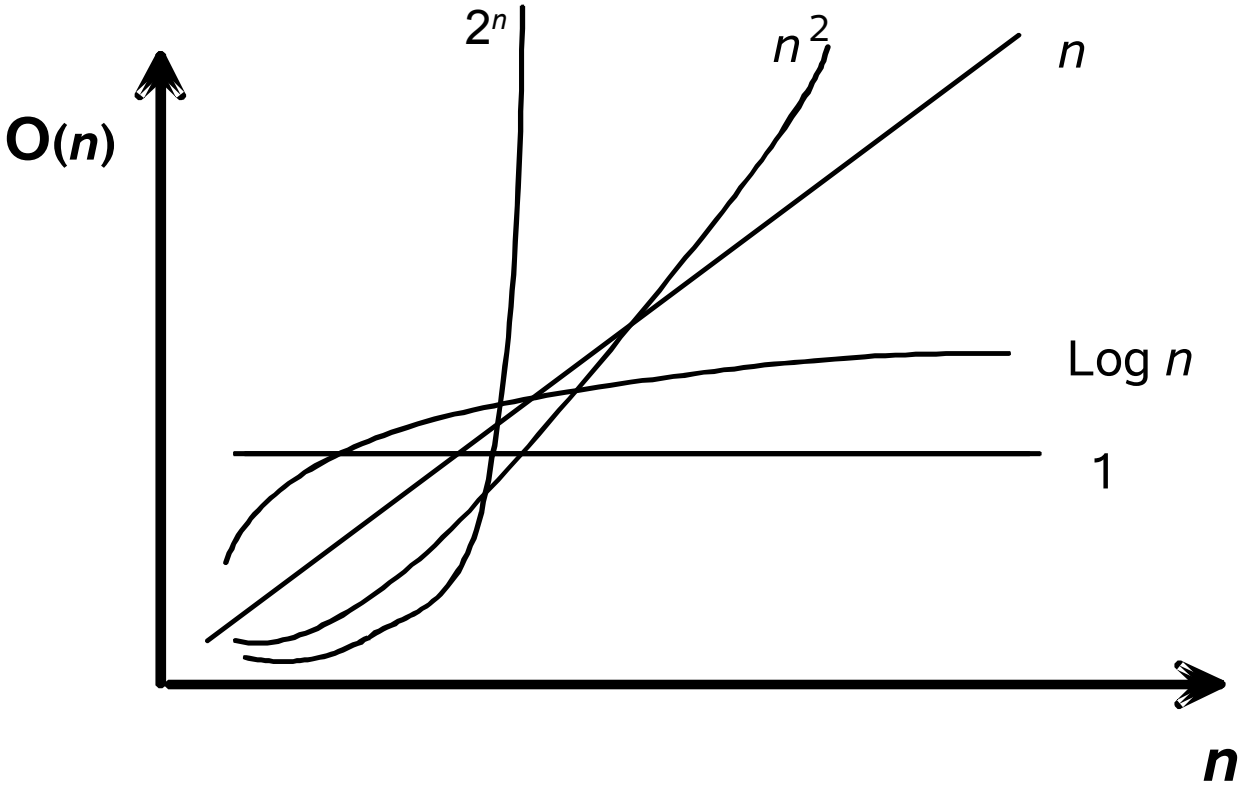
- Définition – complexité d'un algorithme
  - mesure du nombre d'**opérations fondamentales** qu'il effectue sur un jeu de données.
  - exprimée comme une fonction de la taille du jeu de données.

- **Ordre de grandeur**
  - Pas la complexité exacte
    - Comportement **asymptotique**
  - En général, borne supérieure
  - Mais aussi : borne inférieure, moyenne

Classes de complexités

- Rapides / Efficaces
  - **Sub-linéaire** : O(log(n))
  - **Linéaires** : O(n)
  - Autres : O(n log(n))
- Peux efficaces
  - **Polynomiaux :** $O(n^k)$ pour $k > 1$
  - **Exponentiels**
    - Supérieure à tout polynôme en n
    - Impraticables très rapidement

Classe de complexité



Exemple 1 : calculer x^n

- Méthode 1

```
pow1(in réel x, in entierpos n) : réel
    r ← 1
    pour i=1 à n faire
        r ← r*x
    fin pour
    retourner r
```

n *

n+1 ←

…

O(n)

Exemple 1 : calculer x^n

- Méthode 2

```
pow2(in réel x, in entierpos n) : réel
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow2(x,n/2)
        r ← r*r
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

$3 \log_2(n+1) + 1 \leftarrow$

$O(\ln(n))$

Rappels : suite & somme arithmétique

$$\begin{cases} u_0 = a \\ u_{n+1} = u_n + b \end{cases}$$

$$u_n = a + b\,n$$

$$\sum_{i=0}^{n} u_n = \frac{n+1}{2}\left(u_0 + u_n\right)$$

Exemple 1 : calculer x^n

- Méthode 1

```
pow1(in réel x, in entierpos n) : réel
    r ← 1
    pour i=1 à n faire
        r ← r*x
    fin pour
    retourner r
```

n *

n+1 ←

…

Rappels : suite et somme géométrique

$$b \neq 1$$

$$\begin{cases} u_0 = a \\ u_{n+1} = b\, u_n \end{cases}$$

$$u_n = a\, b^n$$

$$\sum_{i=0}^{n} u_n = u_0 \frac{1 - b^{n+1}}{1 - b}$$

Rappel : suite arithmético-géométrique

$$b \neq 1$$

$$\begin{cases} u_0 = a \\ u_{n+1} = b\,u_n + c \end{cases}$$

$$u_n = (a - r)\,b^n + r$$

$$r = \frac{c}{1-b}$$

$$\sum_{i=0}^{n} u_n = (u_0 - r)\,\frac{1-b^{n+1}}{1-b} + (n+1)\,r$$

Complexité moyenne

- But
  - Pas de borne, mais le coût moyen

- Contextes

  (a) Le nombre d'éléments varient

  (b) Le coût dépendent de la valeur des n éléments

- Somme des coûts pondérée par leur fréquence

  (a) $\sum_{n=0}^{n^{max}} p(n)C(n)$

  (b) $\sum_{k=0}^{K} p(k)C(n_k)$

Avantages / inconvénients

- Pire cas
  - Permet de **borner** le coût d'un appel
    - Important pour le « temps-réel »
  - Peut-être éloigné d'un usage courant

- Cas moyen
  - Consommation en **usage courant** (nombreux appels)
  - Le coût d'une exécution peut-être très éloigné
  - Très lié aux hypothèses d'utilisation.

Ex. 2 : « le plus petit élément en tête »

- Hypothèses

  V={v_1,...,v_n} parmi n valeurs distinctes et n ≥ 1

- Algorithme

```
test( in collection V, in entier n) : booléen
    i ← 2
    tant que i <= n et v_i > v_1 faire
        i ← i+1
    fin tant que
    retourner i > n
```

Coût le pire : n-1 itérations

Probabilités en espace fini

- Caractéristiques

Soit X un variable aléatoire
pouvant prendre les valeurs $\{x_1,..,x_K\}$

$$\mathrm{P}\left[X=x_i\right]\geq 0$$

$$\sum_{i=1}^{K}\mathrm{P}\left[X=x_i\right]=1$$

- Espérance / moyenne d'un fonction de $\mathrm{f}(x)$

$$\mathrm{E}\left[\mathrm{f}\left(X\right)\right]=\sum_{i=1}^{K}\mathrm{P}\left[X=x_i\right]\mathrm{f}\left(x_i\right)$$

Dénombrement / analyse combinatoire

- Combinaisons ordonnées de n éléments
  - factorielle

$$n!$$

- Combinaisons ordonnées de k parmi n éléments
  - arrangement

$$A_n^k = \frac{n!}{k!}$$

- Combinaisons de k parmi n éléments
  - coefficient binomial

$$\binom{k}{n} = C_n^k = \frac{n!}{k!(n-k)!}$$

Ex. 2 : « le plus petit élément en tête »

- Hypothèses

  V={v_1,...,v_n} parmi n valeurs distinctes et n ≥ 1

- Algorithme

```
test( in collection V, in entier n) : booléen
    i ← 2
    tant que i <= n et v_i > v_1 faire
        i ← i+1
    fin tant que
    retourner i > n
```

Coût le pire : n-1 itérations Coût moyen

$$\frac{n-1}{n}+\sum_{k=1}^{n-1}(n-k)\frac{(k-1)!}{n!}\left(\sum_{2}^{k+1}(i-1)\frac{(n-i)!}{(k-i+1)!}\right)$$

La récursivité

- Fonction/procédure s'appelle elle-même
  - Approche simple et naturelle pour les suites
    - Une **condition d'arrêt**
    - Un appel avec une **complexité diminuée**
- Arbre des appels
  - Liste des appels connectés
- **Pile** des appels
  - Récursion → **sauvegarder** le contexte courant
  - Parcours en profondeur du graphe

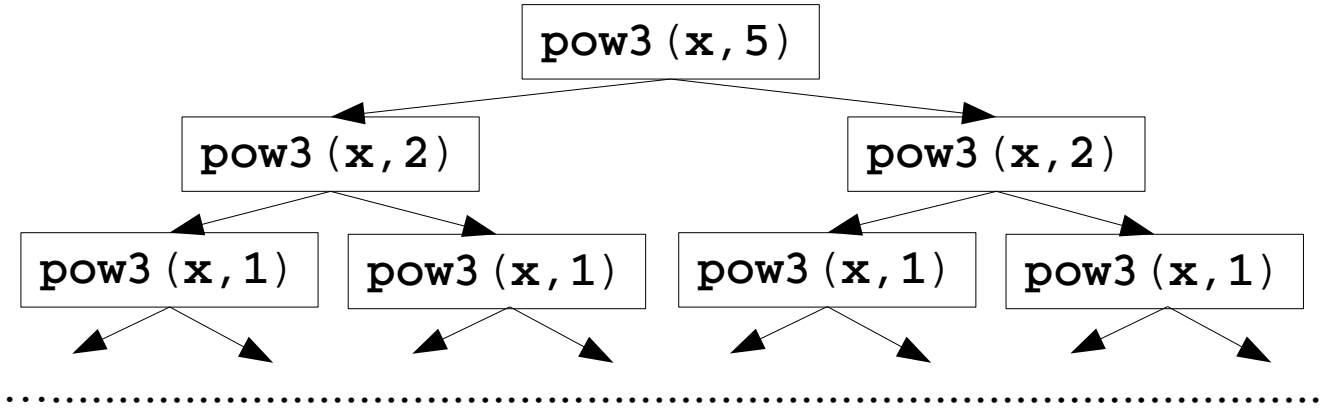
Exemple 3 : pile / appels pour le calcul de x^n

- Méthode 3

```
pow3(in réel x,in entier n): réel
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow3(x,n/2)*pow3(x,n/2)
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

O(n)

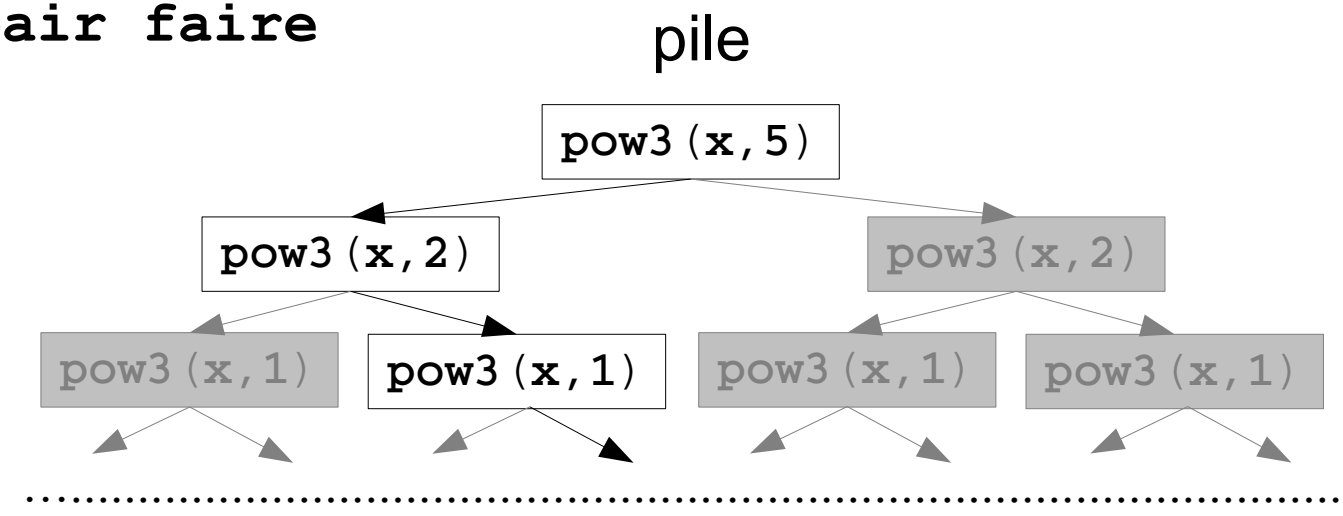
appels



Exemple 3 : pile / appels pour le calcul de x^n

- Méthode 3

```
pow3(in réel x,in entier n): réel
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow3(x,n/2)*pow3(x,n/2)
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

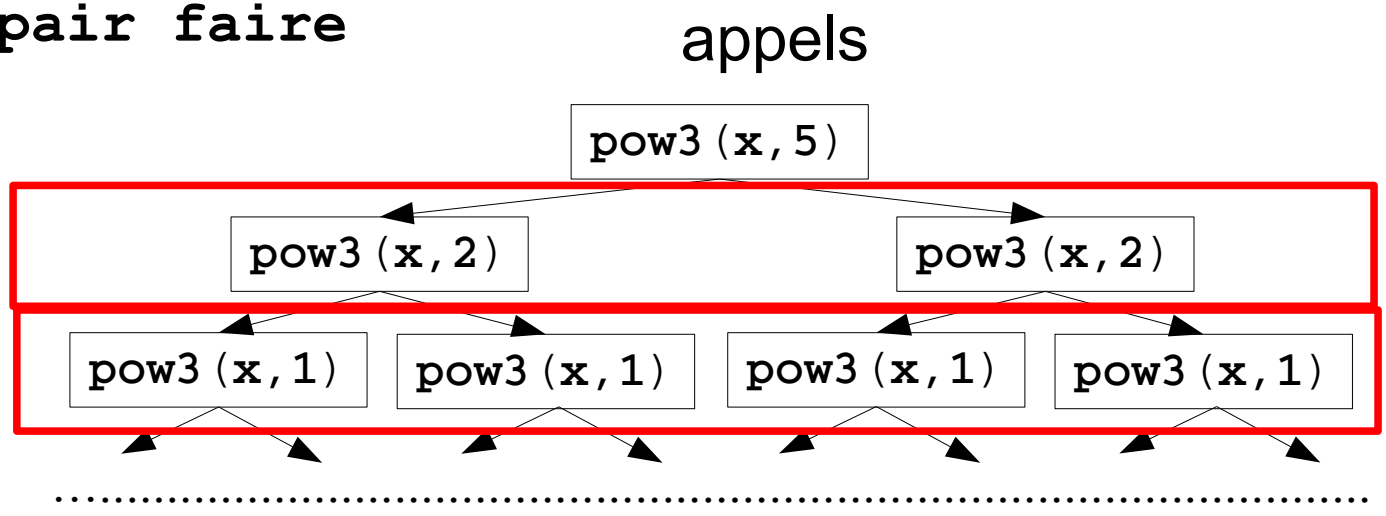


Coût mémoire = O(ln(n)) pour sauvegarder les appels

Exemple 3 : pile / appels pour le calcul de x^n

- Méthode 3

```
pow3(in réel x,in entier n): réel
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow3(x,n/2)*pow3(x,n/2)
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

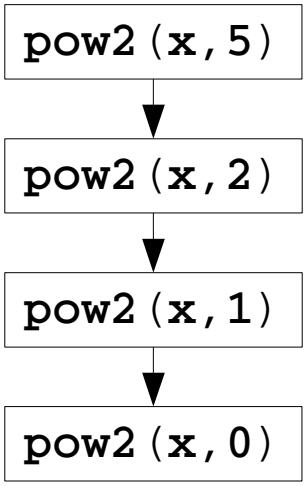


Exemple 3 : pile / appels pour le calcul de x^n

- Méthode 2

```
pow3(in réel x,in entier n): réel
    si n = 0 faire
        r ← 1
    sinon faire
        r ← pow3(x,n/2)
        r ← r*r
        si n est impair faire
            r ← r*x
        fin si
    fin si
    retourner r
```

Appels / pile



Coût calcul = Pile = O(ln(n))

Itération (rappel) : boucles / parcours

- Buts :
  - Répéter un **même action**
  - **Parcourir** une collection d'élément de même type
- 3 types
  - ***tant que*** X *faire* Y :
    - exécuter les actions Y tant X est vérifiée
  - *faire* Y ***jusqu'à*** X
    - exécuter les actions Y jusqu'à ce que X soit vérifiée
  - ***Pour*** i=début *à* fin *faire Y*
    - exécuter les actions Y pour i prenant les valeurs de début à fin

Terminaison d'un algorithme

- Récursif
  - Condition d'arrêt
  - La complexité décroît vers la condition d'arrêt

- Itératif
  - Condition d'arrêt
  - La valeur testée change à chaque itération et temps vers la condition d'arrêt

Réduire la complexité

René Descartes dans le *Discours de la Méthode* :

> « *diviser chacune des difficultés que j'examinerois, en autant de parcelles qu'il se pourroit, et qu'il seroit requis pour les mieux résoudre* »

- **Diviser pour régner**

  On divise le problème en plusieurs sous problèmes et on combine les résultats pour obtenir la solution globale.

Autres types d'algorithmes

- Algorithmes exhaustifs
  - Tester tous les cas
  - Lorsque l'on ne sait rien faire d'autre

- Algorithmes gloutons
  - Pas à pas
  - Ne pas revenir en arrière
  - Ex : optimisation

Algorithmique
&
Structures de donnée

INSTITUT d'OPTIQUE GRADUATE SCHOOL
ParisTech

Merci à Maxime CROCHEMORE

Extension : notion d'efficacité

- Avant de coder
  - Définir l'algorithme
  - Prévoir son coût en calcul
  - Prévoir son coût en mémoire

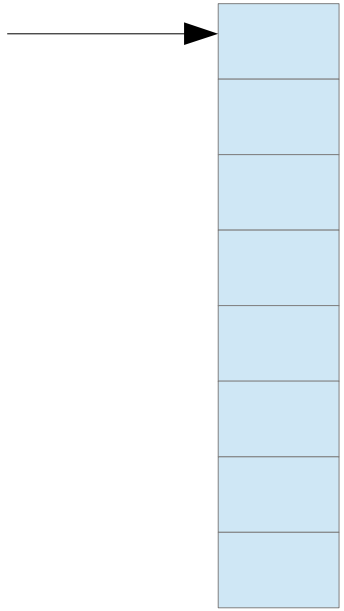
- Calibration
  - Choix des structures
  - Choix de l'algorithme

Importance des structures de donnée

- La manière de représenter impacte
  - L'occupation mémoire
  - Le coût d'accès
    - Ex pour un ensemble : accéder au k^{ème} élément
  - Le coût de manipulation
    - Ex pour un ensemble : ajouter/supprimer un élément

Exemple : structure de tableau

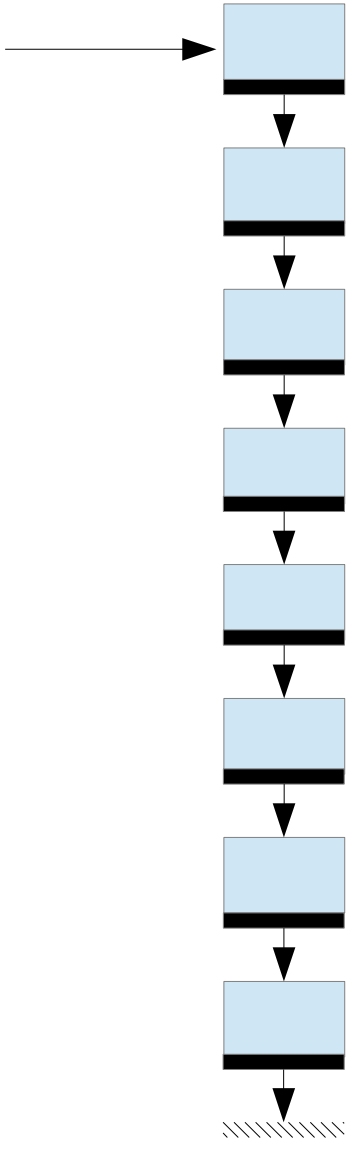
- Mémoire contiguë
  - Coût mémoire : n*taille des éléments
- Coût accès au k^{ième} élément
  - constant = O(1)
- Coût d'insertion après le k^{ième} élément
  - O(n-k), si pas de re-allocation
  - O(n) sinon
- Coût d'insertion avant le k^{ième} élément
  - idem



http://www.cplusplus.com/reference/stl/vector/operator%5B%5D/

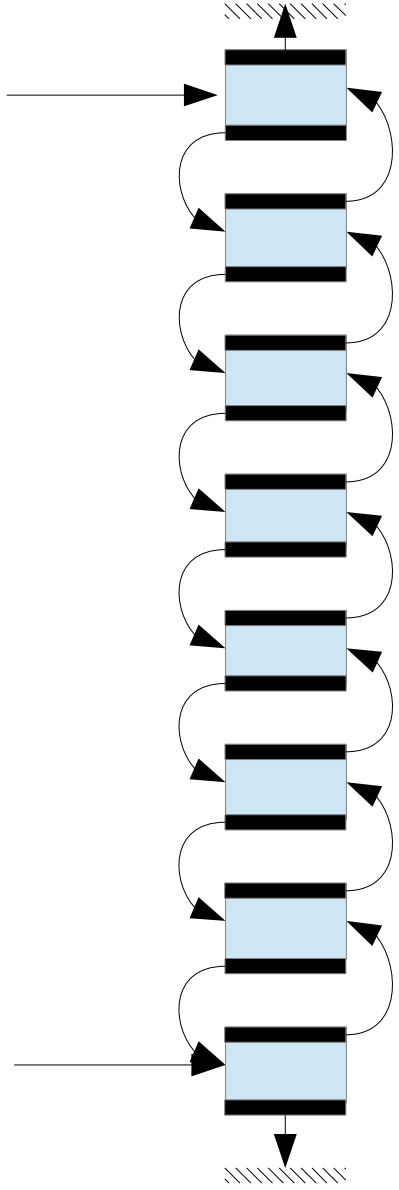
Collections : liste simplement chaînée

- Mémoire non-contiguë
  - Coût mémoire : n*(type+références)
- Coût accès au k^{ième} élément
  - linéaire = O(k)
- Coût d'insertion après le k^{ième} élément
  - linéaire = O(k) si on part du début
  - constant si on connaît la position du k^{ième} élément
- Coût d'insertion avant le k^{ième} élément
  - O(k)



Collections : liste doublement chaînée

- Mémoire non-contiguë
  - Coût mémoire : n*(type+2*références)
- Coût accès au k^{ième} élément
  - linéaire = O(k)
- Coût d'insertion après le k^{ième} élément
  - linéaire = O(k) si on part du début
  - constant si on connaît la position du k^{ième} élément
- Coût d'insertion avant le k^{ième} élément
  - idem



Exercice : Liste

- Utiliser un tableau pour implémenter une liste
  - Hypothèse, le nombre maximale est bornée par N

Un pointeur n'est rien d'autre qu'un indice dans un tableau !

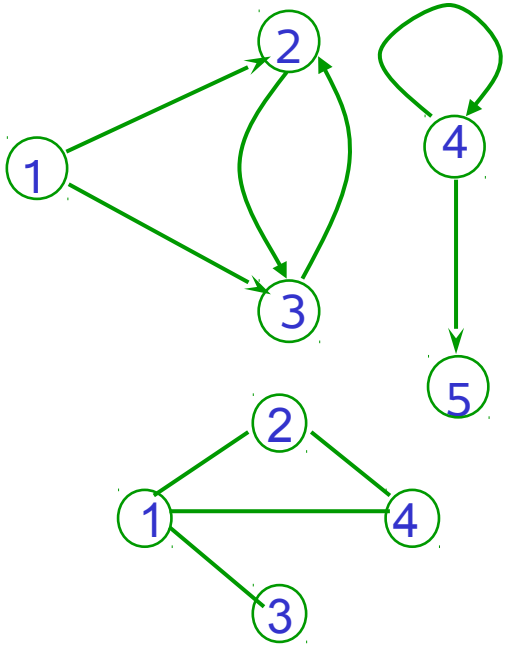
- Actions à faire sur une liste : complexité
  - Concaténer deux listes
  - Connaître sa taille

O(1)

O(n)

Graphe

- Définition
  - Des **nœuds** reliés par des **arêtes**
    - Nœuds **adjacents** = il sont reliées
  - Deux catégories
    - Non-orienté
    - Orienté
- Actions
  - Parcours (aller d'un nœud à son(ses) successeur(s))
  - Ajouter / supprimer une arête
  - *Ajouter / supprimer un nœud*



Graphe : représentation 1

- **Ensemble** N de k nœuds
- **Matrice** M de k² adjacences :
  - M[i,j]=vrai ↔ N[j] est relié à N[i]
  - Non orienté : M est symétrique
- Coûts
  - Mémoire : O(k²)
  - Accéder aux éléments suivant d'un nœud : O(k)
  - Ajouter une arête : O(1)

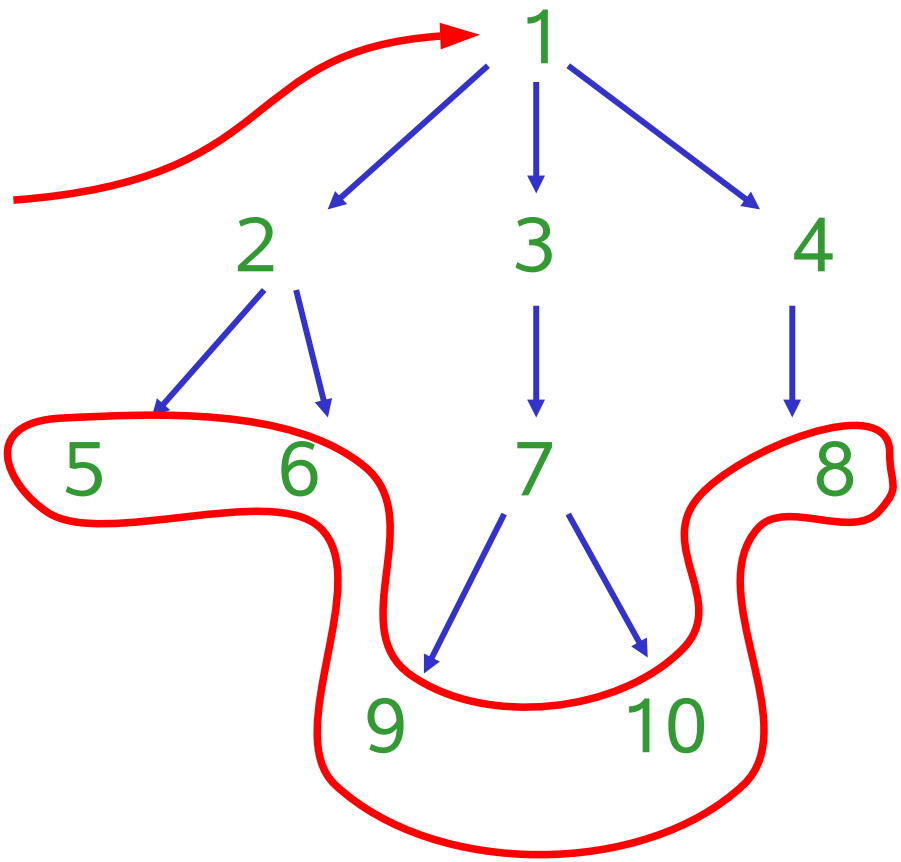
Graphie : représentation 2

- Pour chaque nœud
  - L'ensemble des successeurs

- Coûts
  - Mémoire : O(k+a) (a = le nombre d'arête)
    - $a \leq k^2$
    - Entre O(k) et O(k^2)
  - Accéder aux éléments suivant d'un nœud : O(k)
    - Raffiner en O(a^{max})
  - Ajouter une arête : O(1)

Arbre

- Définition
  - **Graphe non-orienté sans cycle**
  - **Racine** : nœud de départ
  - **Feuilles** : derniers nœuds
- Actions
  - Ajouter / supprimer d'un nœud
  - Parcours en profondeur
    - 1, 2 , 5, 6, 3, 7, 9, 10, 4, 8
  - Parcours en largeur
    - 1, 2 , 3, 4, 5, 6, 7, 8, 9, 10



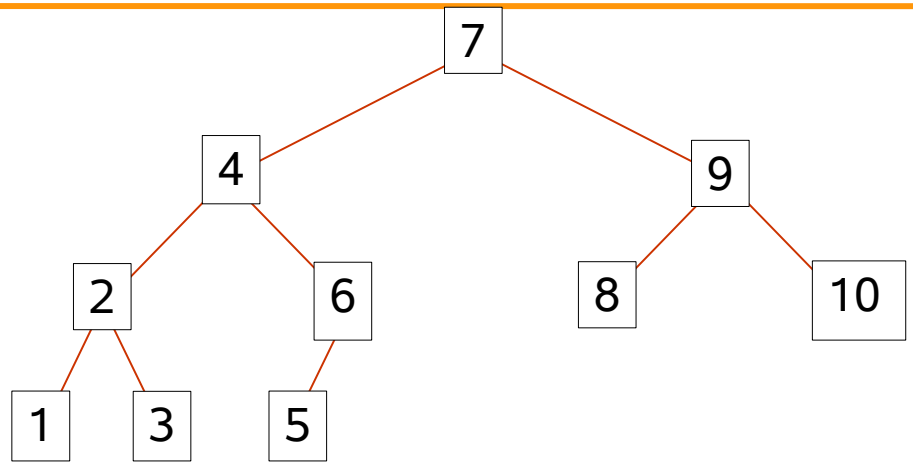
Parcours d'un arbre binaire

- **Préfixé**
  - Nœud, fils gauche puis droit

    7 4 2 1 3 6 5 9 8 10
- **Postfixé**
  - Fils gauche puis droit, nœud

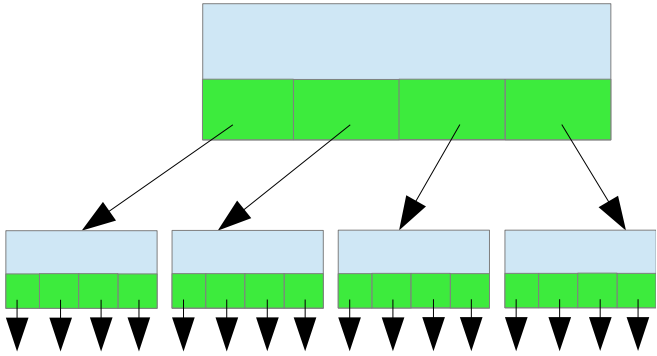
    1 3 2 5 6 4 8 10 9 7
- **Infixé**
  - Fils gauche, nœud , fils droit

    1 2 3 4 5 6 7 8 9 10



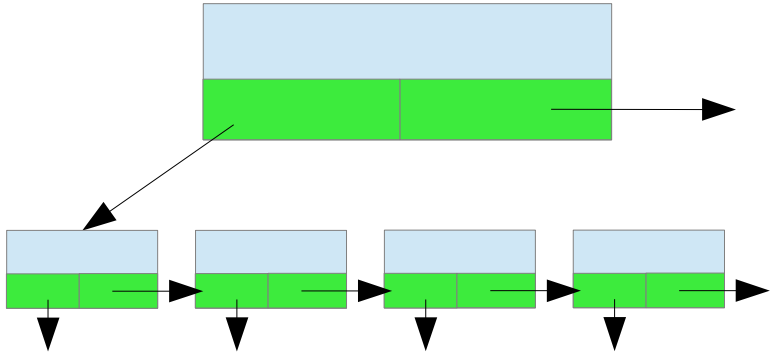
Arbre : représentations

- Solution 1 (classique)
  - Chaque nœud : les fils



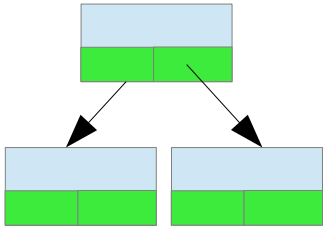
- Solution 2
  - Chaque nœud : le premier fils + plus frère suivant

- Feuille = pas de fils

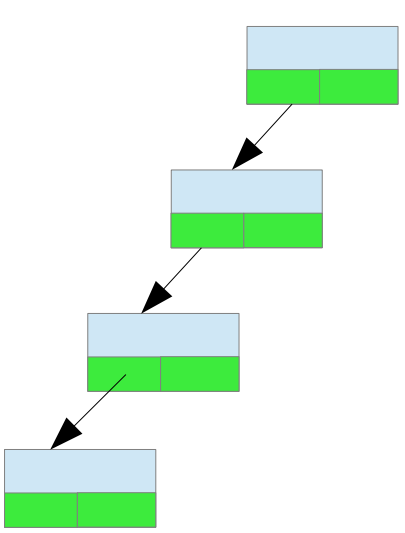


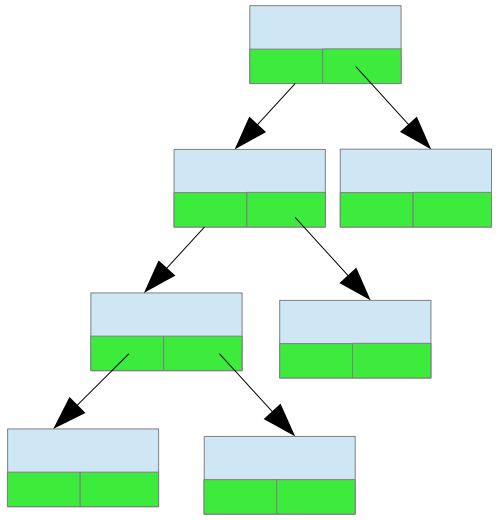
Arbres binaires

- Maximum 2 fils
  - 0 pour une feuille
  - 1 ou 2 pour les autres



- Hauteur de l'arbre ?
  - De ln(n) à n
  - Arbres dégénérés





peigne

Arbre binaire de recherche (ABR)

- Définition
  - Arbre binaire
  - Valeur du nœud
    - supérieure au fils gauche
    - inférieure au fils droit

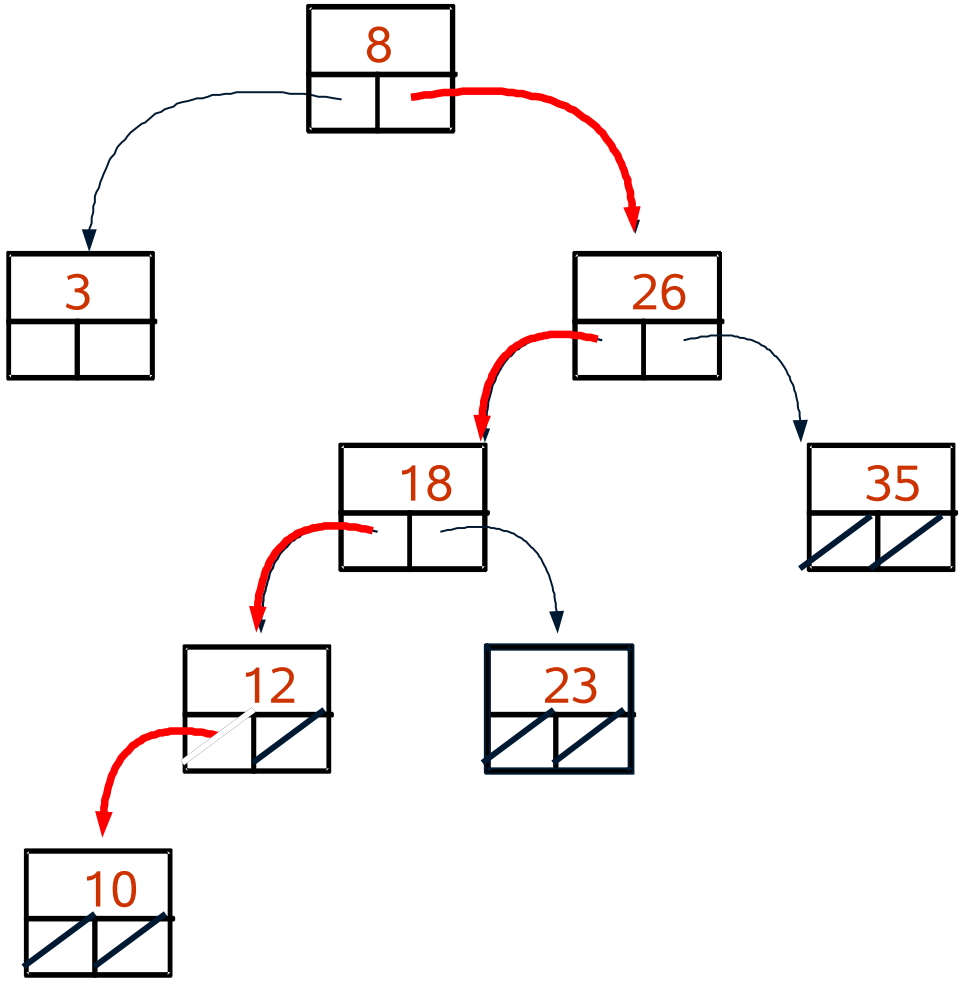
- Actions principales
  - Ajout / suppression en préservant la propriété

ABR : Ajout d'un élément

Nouveau nœud = feuille

1. Chercher le père du nouveau nœud
   - À gauche si plus petit
   - À droite si plus grand
2. En fin de parcours (feuille)
   - Chaîner

ABR : Ajout d'un élément (10)



ABR : Ajout d'un élément (itératif)

```
nœud ← racine de abr
père ← vide
Tant que nœud n'est pas vide faire
  père ← nœud
  Si val < valeur de nœud faire
    nœud ← fils gauche de nœud
  Sinon
    nœud ← fils droit de nœud
  Fin si
Fin tant que
…..
```

ABR : Ajout d'un élément (itératif)

```
……
nœud ← créer un noeud contenant val
si père est vide faire
  racine de abr ← nœud
sinon faire
  Si val < valeur de père faire
    fils gauche de père ← nœud
  Sinon faire
    fils droit de père ← nœud
  Fin si
Fin si
```

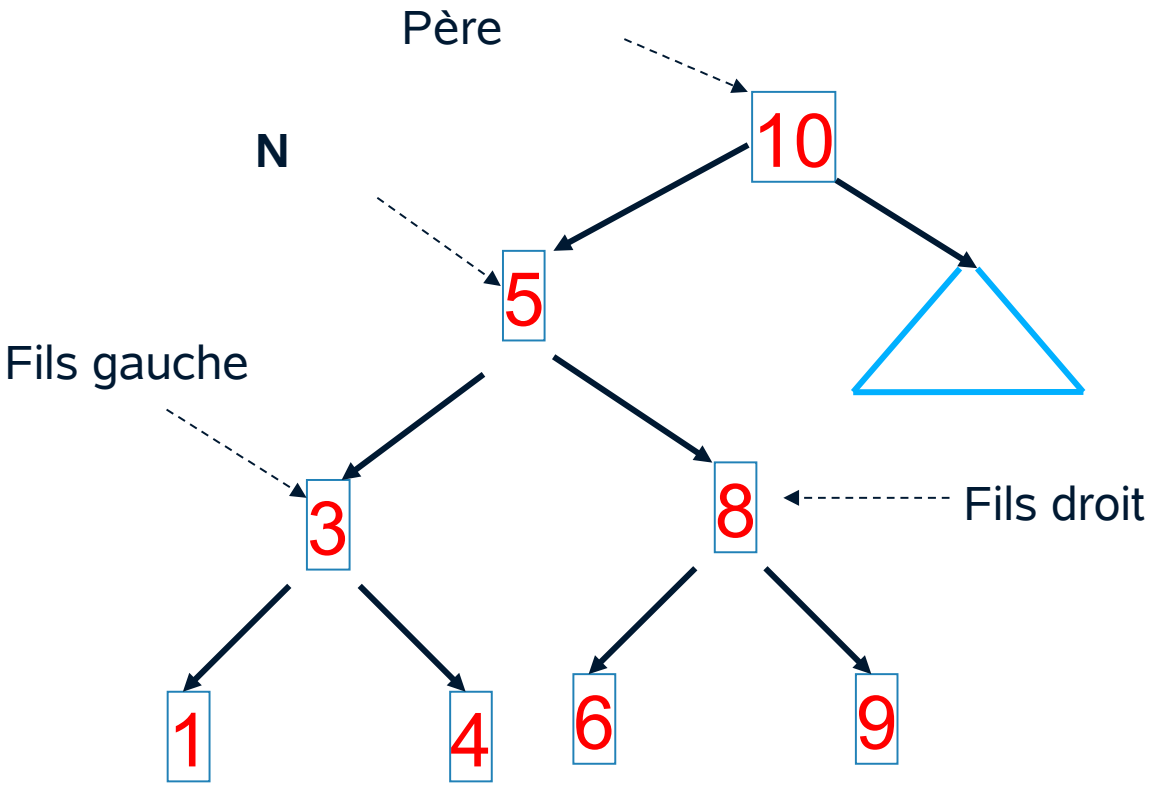
Coût O(n)

ABR : suppression d'un élément

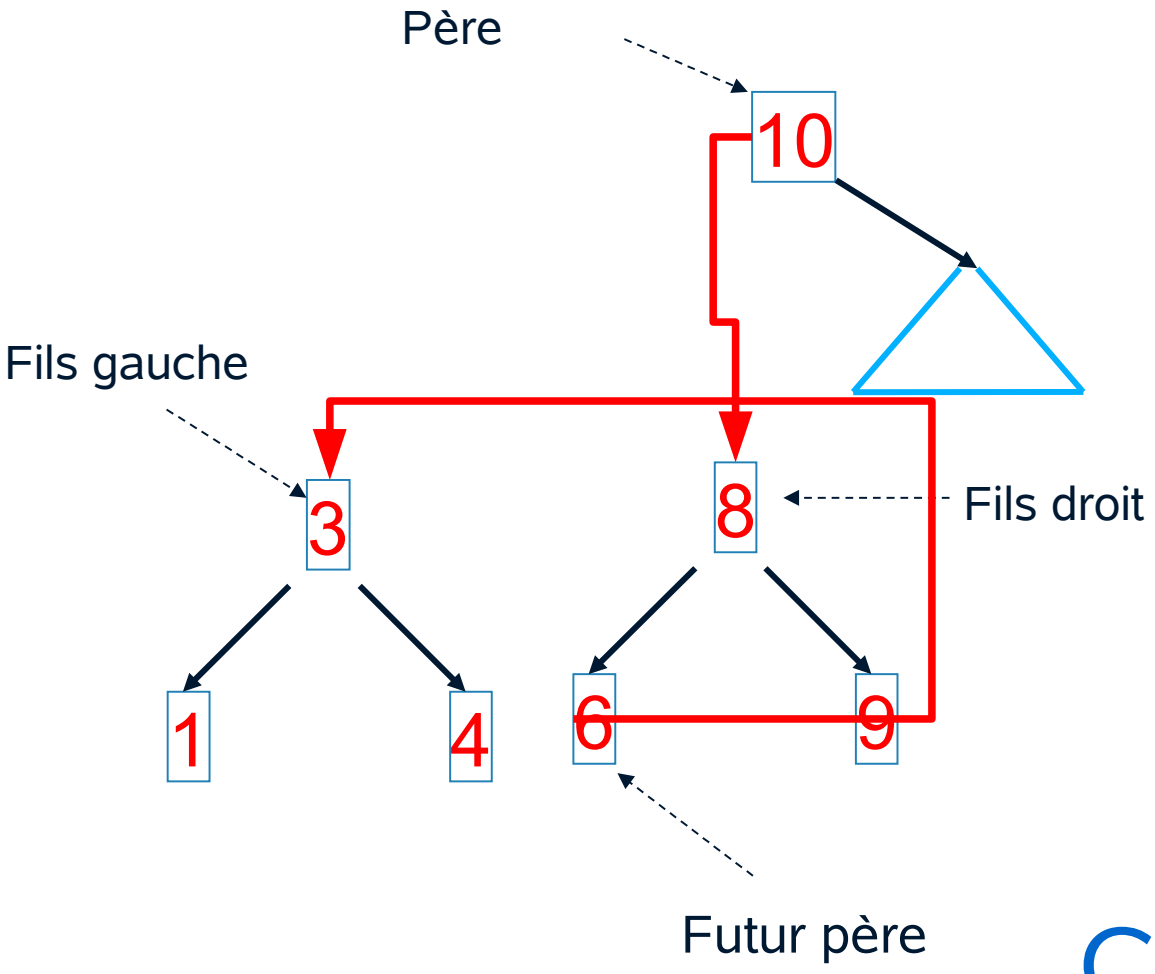
Soit **N** le nœud à supprimer

1. Chercher le père de **N**
2. Chercher, dans le sous-arbre droit du nœud, le futur père **P** du premier nœud sans fils gauche
3. Attacher le sous-arbre gauche de **N** à **P**
4. Attacher le fils droit de **N** à son père
5. Traiter les cas particuliers
   - pas sous-arbre droit
   - suppression de la racine
   - ….

ABR : suppression d'un élément (5)



ABR : suppression d'un élément (5)



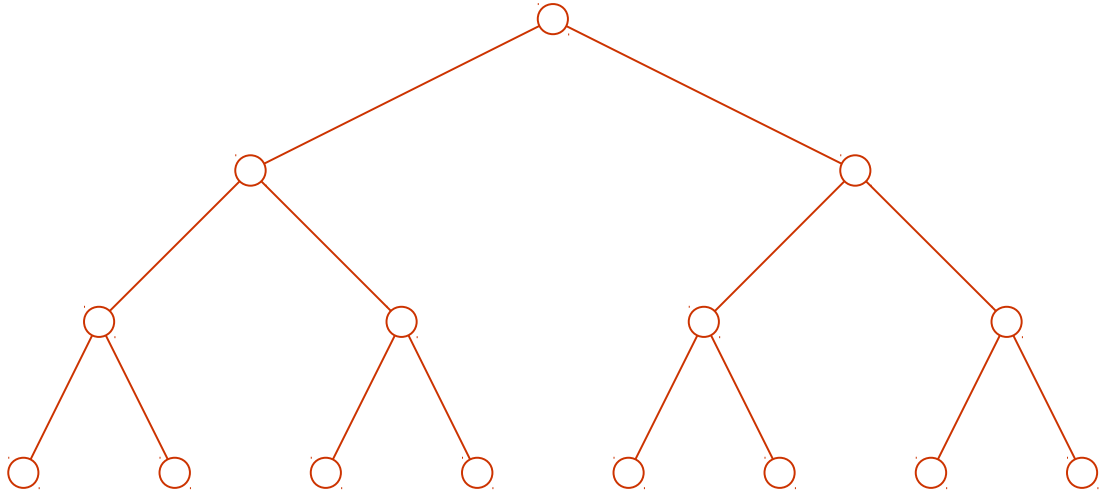
Coût O(n)

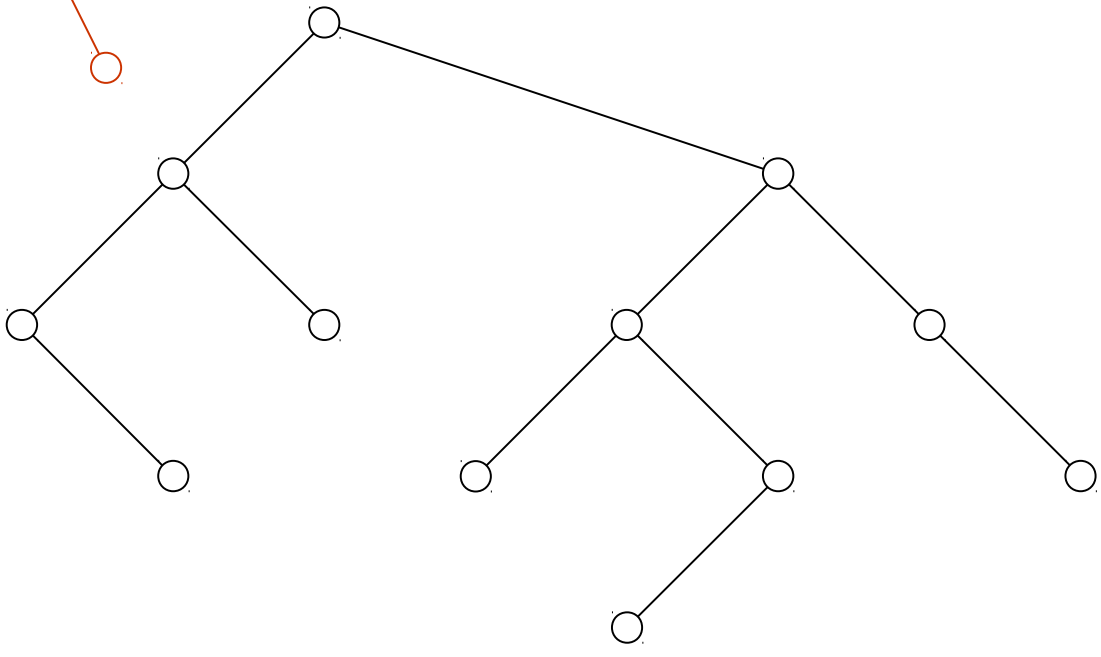
Arbre binaire de recherche équilibré (AVL)

Adelson – Velskii – Landis

- Principale problème
  - Arbres dégénérés → coût linéaire
- Solution
  - Éviter la dégénérescence
- Définition :
  - Arbre binaire de recherche
  - Profondeur ne diffère d'au plus 1
    - entre sous-arbre gauche et droit
  - Hauteur total $\log_2(n)$

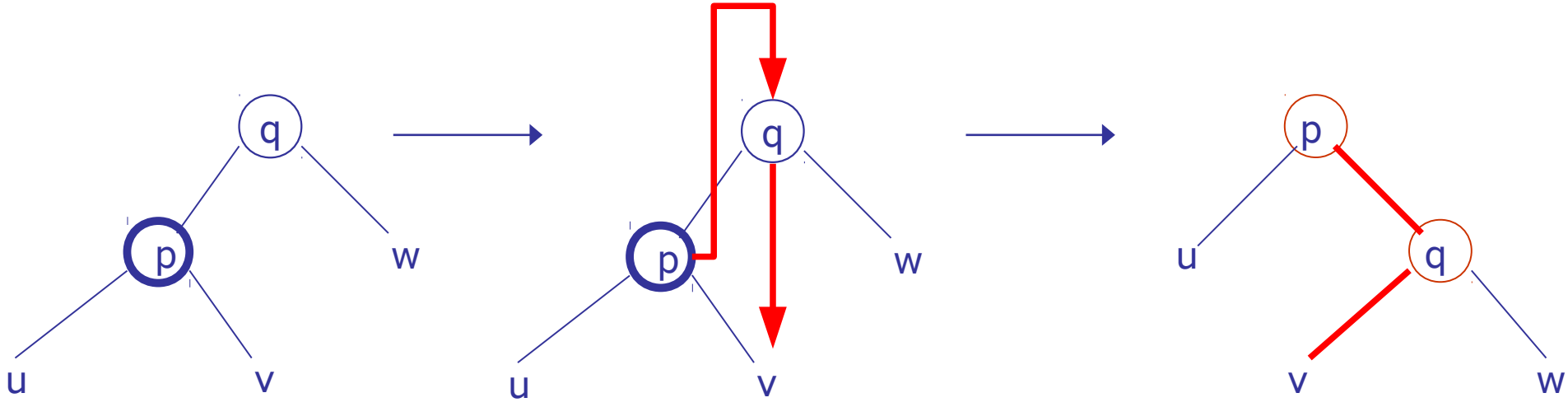
AVL : exemples



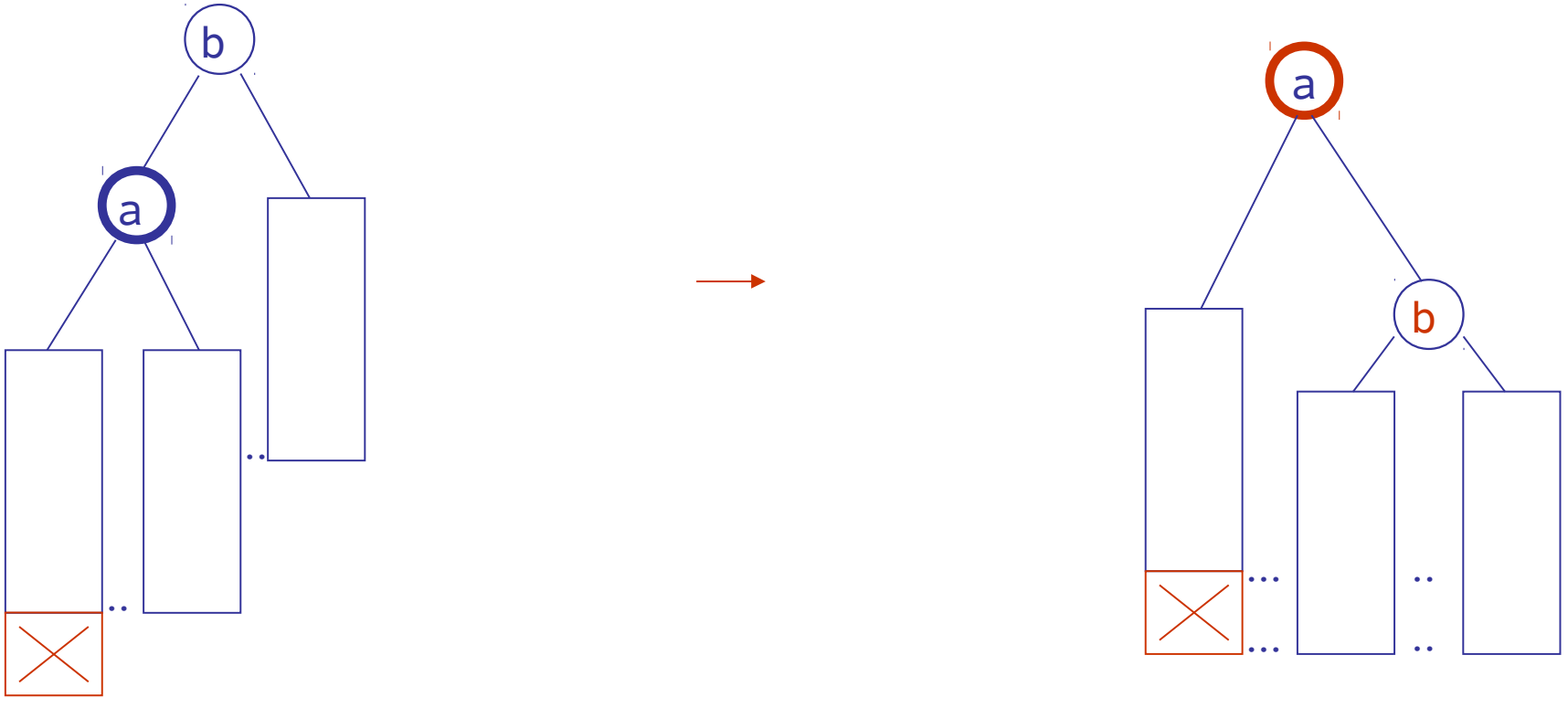


AVL : adaptation des algorithmes

- Ajouter et Supprimer
  - Suivi d'un phase d'équilibrage
- Équilibrage : rotations successives
  - Sous-arbre gauche trop grand

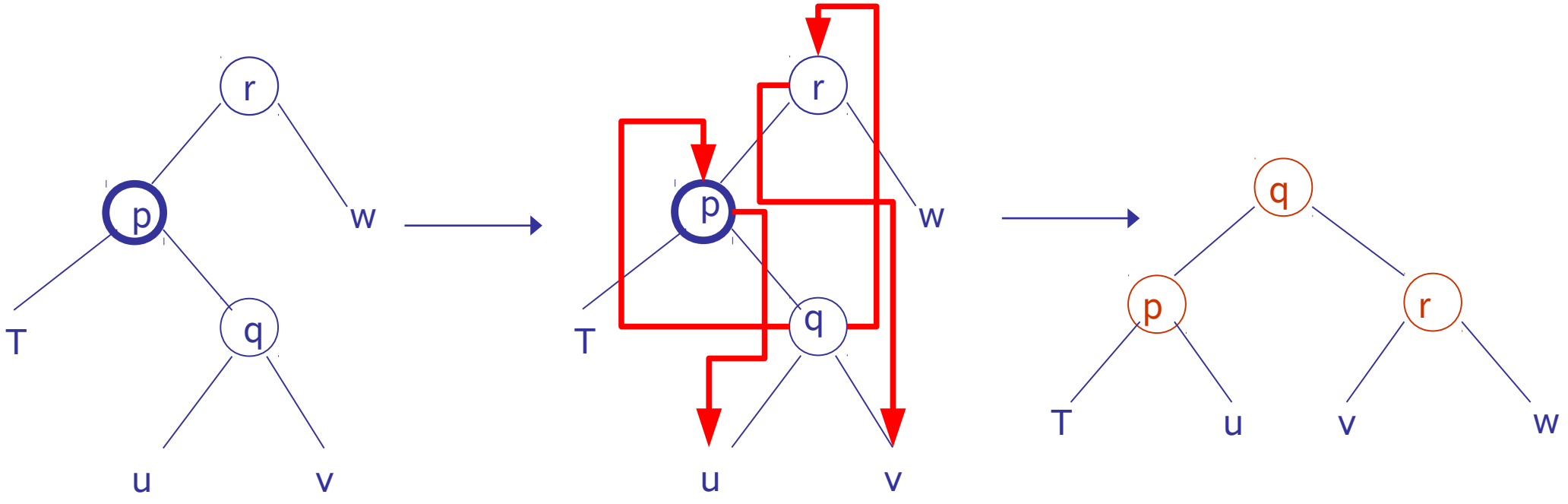


Rotation droite

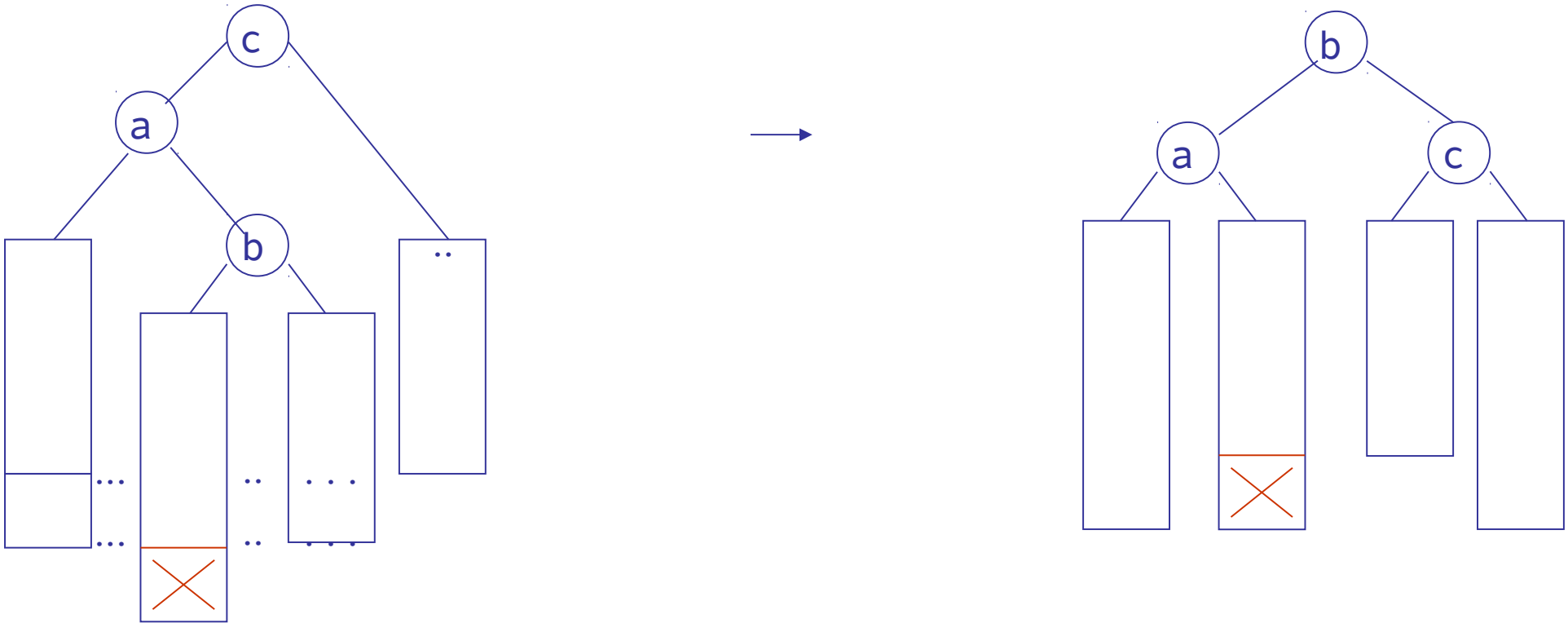


AVL : adaptation des algorithmes

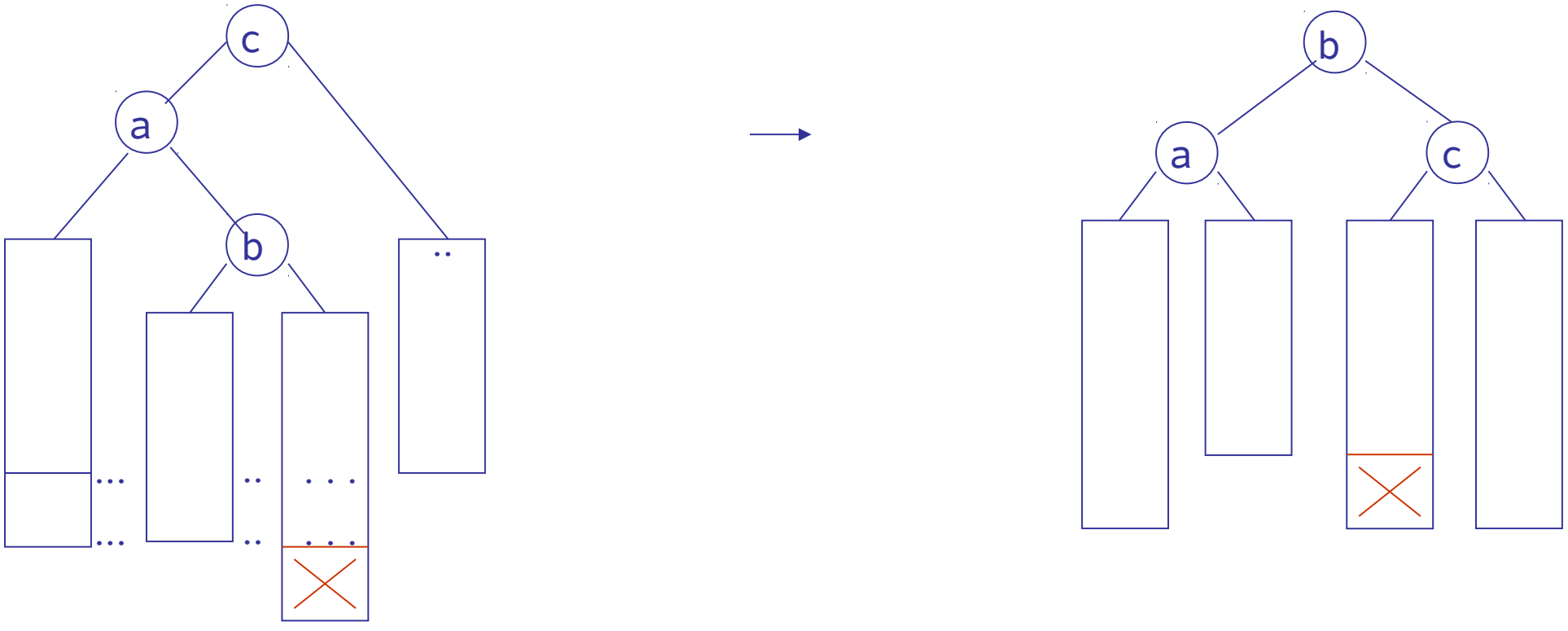
- Ajouter et Supprimer
  - Suivi d'un phase d'équilibrage
- Équilibrage : rotations successives
  - Sous-arbre droit trop grand



Rotation gauche-droite



Rotation gauche-droite



AVL : Coût Insertion / Suppression

- Trouver la position

$O(\ln(n))$

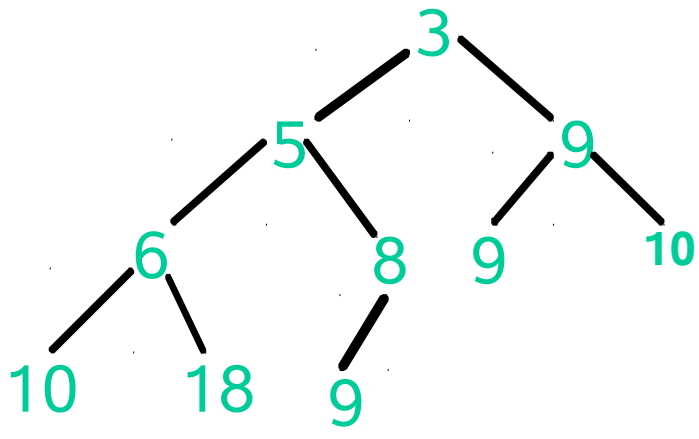
- Équilibrage successifs
  - En remontant, il peut y en avoir plusieurs

$O(\ln(n))$

Arbre parfait partielle ordonnée : le tas

- Définition :
  - Arbre binaire
  - Parfait : tassé à gauche
  - Partiellement ordonné
    - Valeur père ≥ valeurs fils

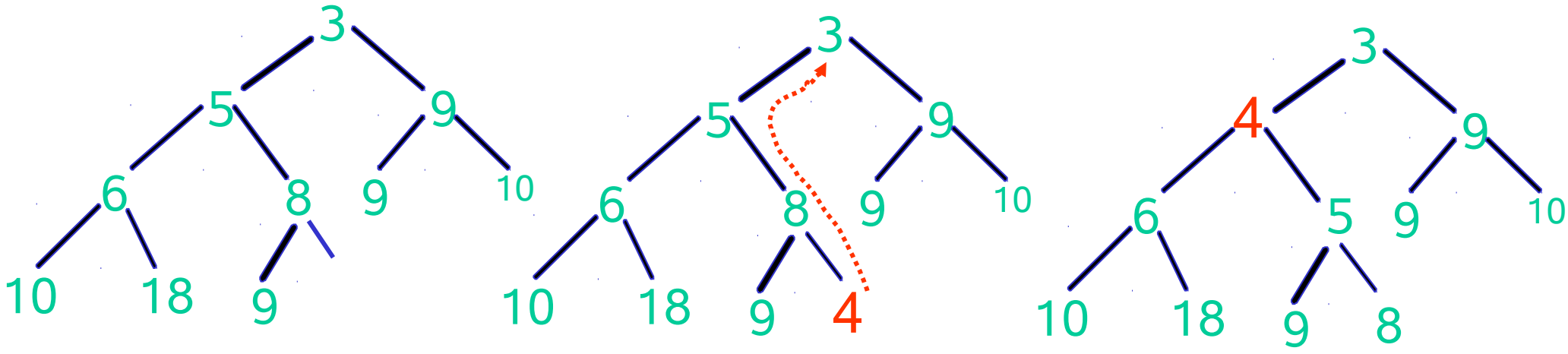
      ou
    - Valeur père ≤ valeurs fils



Hauteur = $\log_2 n$

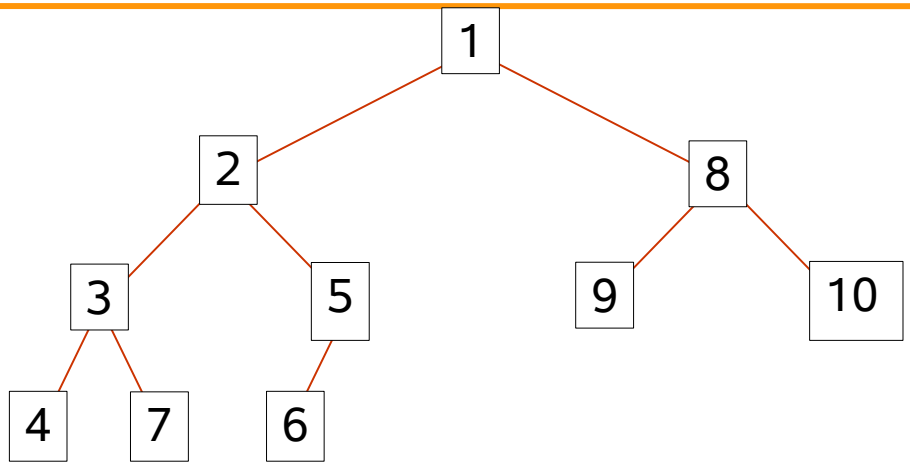
Ajouter

- Principe
  - Ajouter dans la feuille la plus à gauche
  - Restaurer l'ordre partiel



Le tas

- Avantage
  - Très compacte
  - Stockage dans un tableau T
    - Racine T[1]
    - Fils gauche de T[i] = T[2i]
    - Fils droit de T[i] = T[2i+1]



1|2|8|3|5|9|10|4|7|6

- Désavantage
  - Si tableau, pas dynamique (ou taille bornée)

Ajouter : tas dans un tableau

```
Ajouter (inout tas t, in taille n, in élément x)
  n ← n+1
  i ← n
  t[n] ← x
  Tant que i > 0  et t[i] < t[(i-1)/2] faire
    t[i] ↔ t[(i-1)/2]
    i ← (i-1)/2
  Fin tant que
```

O(ln(n))

Supprimer le minimum (maximum) dans un tas

```
SupprMin (t tas, n taille > 0)
  n ← n-1
  d ← n
  t[0] ←  t[d+1]
  i ← 0 ;
  Faire
    fin ← vrai ;
    Si  2i+2 <= d alors faire
      Si t[2i+1] < t[2i+2] alors faire
        k ← 2i+1
      Sinon  faire
        k ← 2i+2
      Fin si
      Si t[i] > t[ k] alors faire
        t[i] ↔ t[ k]
        i ← k
        fin ← faux
      Fin si
    Sinon si 2i+1=d et t[i] > t[ k] faire
        t[i] ↔ t[ k]
    Fin si
  Jusqu'à fin
```

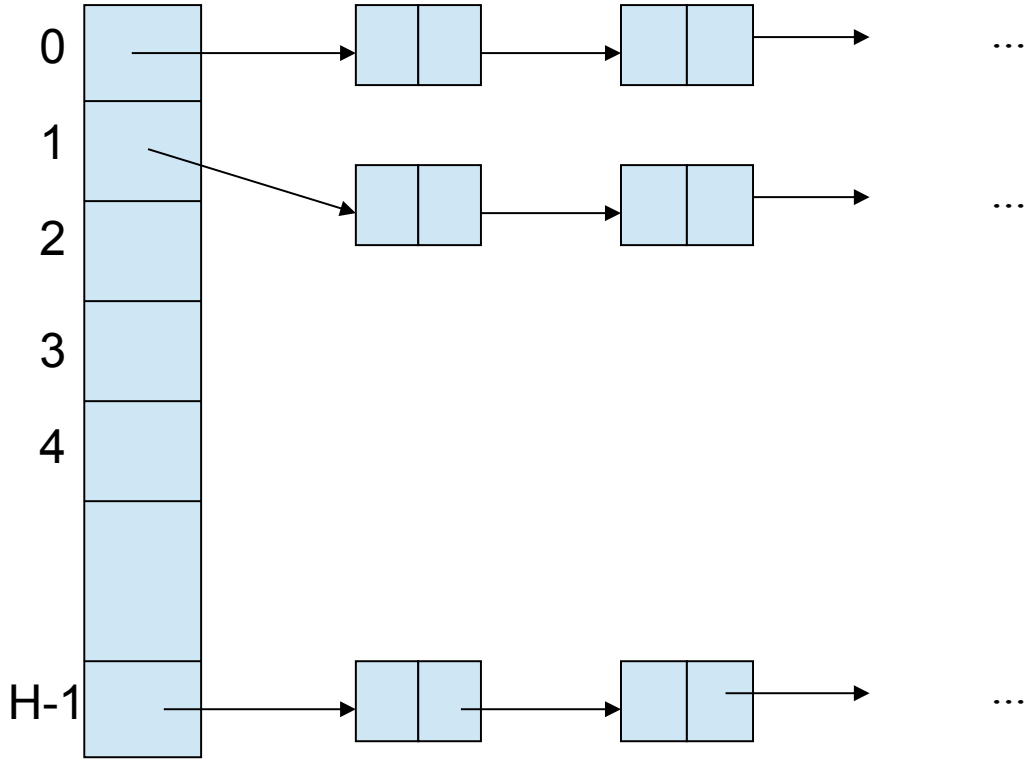
O(ln(n))

Table de Hachage (hash code)

- Définition

  Utilisation d'une clef pour indexer les valeurs

- Avantage
  - Division des coûts de recherche, ajout, suppression par H



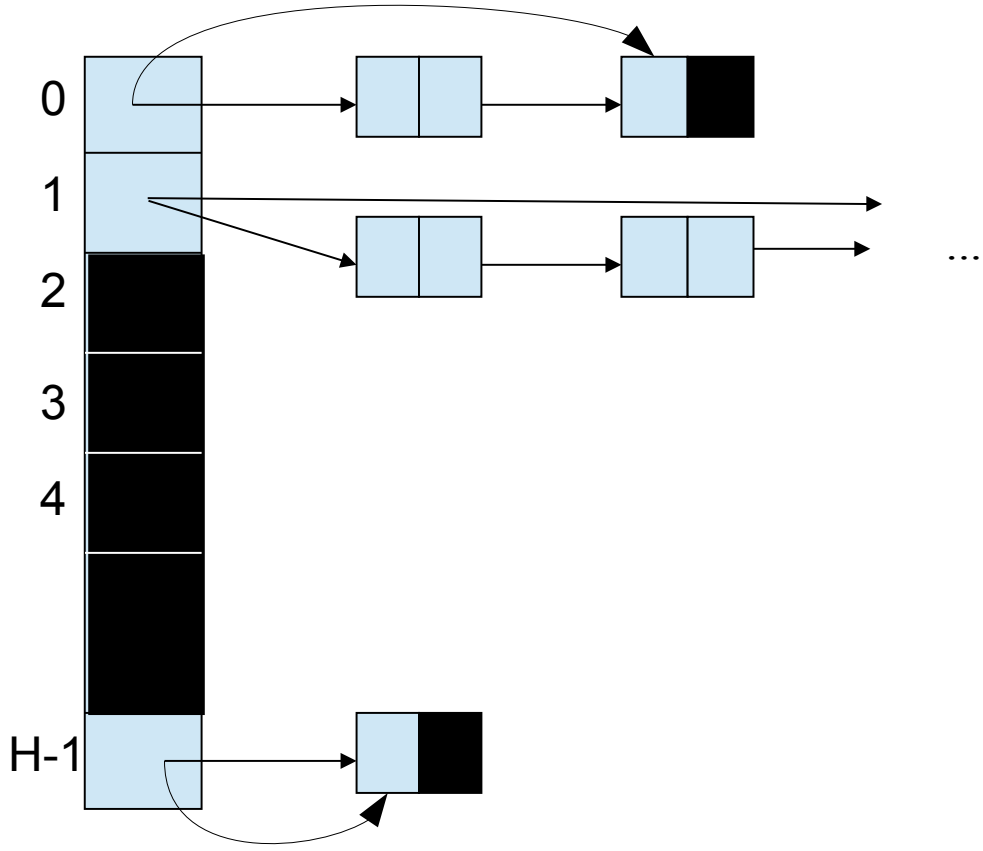
Exemple pour des listes

Autres notions : les files d'attentes

- Contexte
  - **Ajout / Suppression** d'éléments en **continue**
- Types de files classiques
  - **FIFO** : First In, First Out
    - Insertion en queue O(1)
    - Suppression en tête O(1)
  - **LIFO** : Last In, First Out
    - Insertion / Suppression en tête O(1)
  - Avec **priorité**

File d'attente avec priorité

- Implémentation simple : table de hachage
  - File vide : O(H)
  - Insertion LIFO : O(1)
  - Insertion FIFO : O(1)
  - Mémoire : O(H)+O(n)
  - Nombre fini



File d'attente avec priorité infinie

- Utilisation de structure dynamiques
  - Listes / AVL

- Coûts
  - File vide : O(1)
  - Ajout : O(H) / O(ln(H)) avec H le nombre de priorité
  - Mémoire : O(H)+O(n)

Tableau récapitulatif : coût des opérations

	Appartenance	Insérer	Supprimer	Minimum
Tableau	n	n	n	n
Tableau trié	ln(n)	n	n	1
Liste	n	1	1	n
Liste triée	n	1	1	1
ABR	de ln(n) à n	1	1	de ln(n) à n
AVL	ln(n)	ln(n)	ln(n)	ln(n)
Hash code - Liste	n	n	n	n
Hash code - ABR	ln(n)	1	1	1

Le coût d'insertion et suppression se fait sans la recherche de la position. Pour le coût total, il faut donc rajouter le temps d'appartenance.