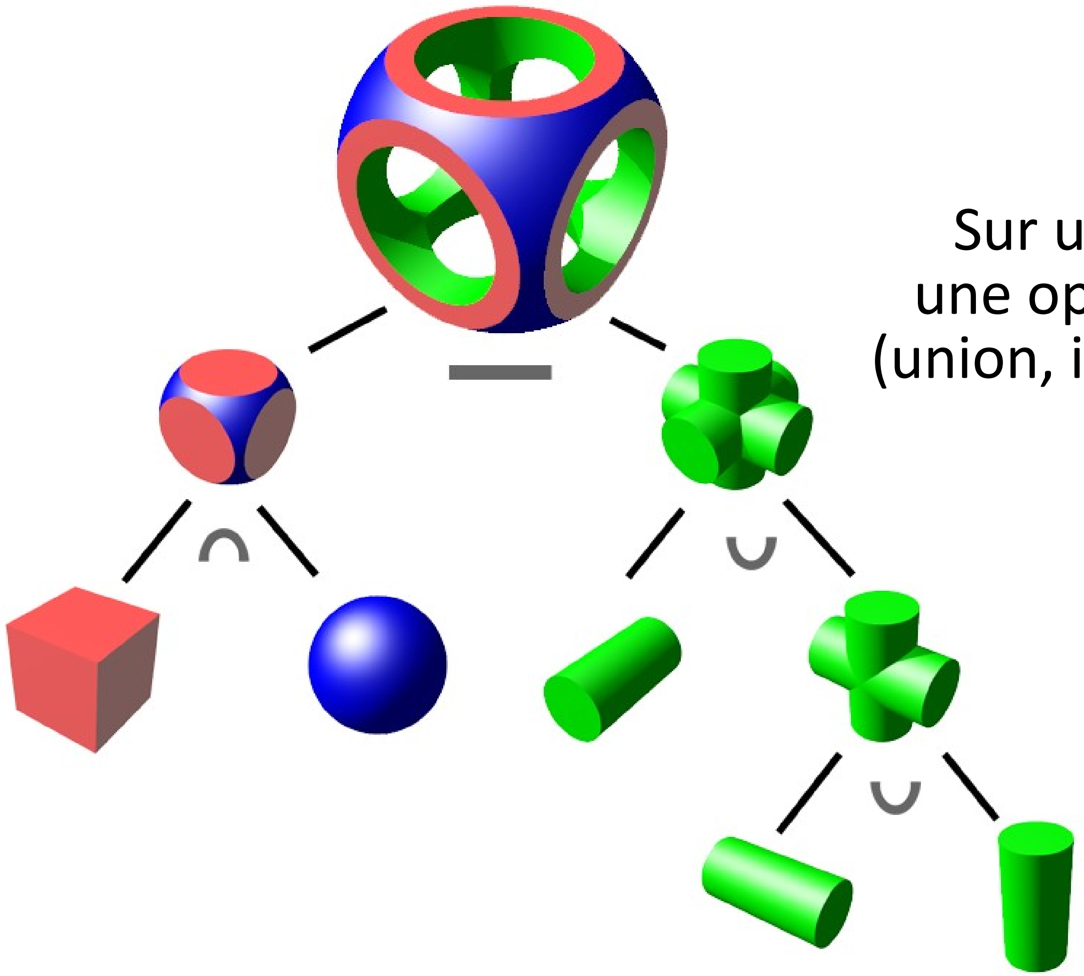
PRALG séance 4

Petit arboretum 2e partie : Stockage et recherche d'information efficaces

Pascal Monasse / Renaud Marlet
Laboratoire LIGM-IMAGINE

Exemple : (dé)composition géométrique

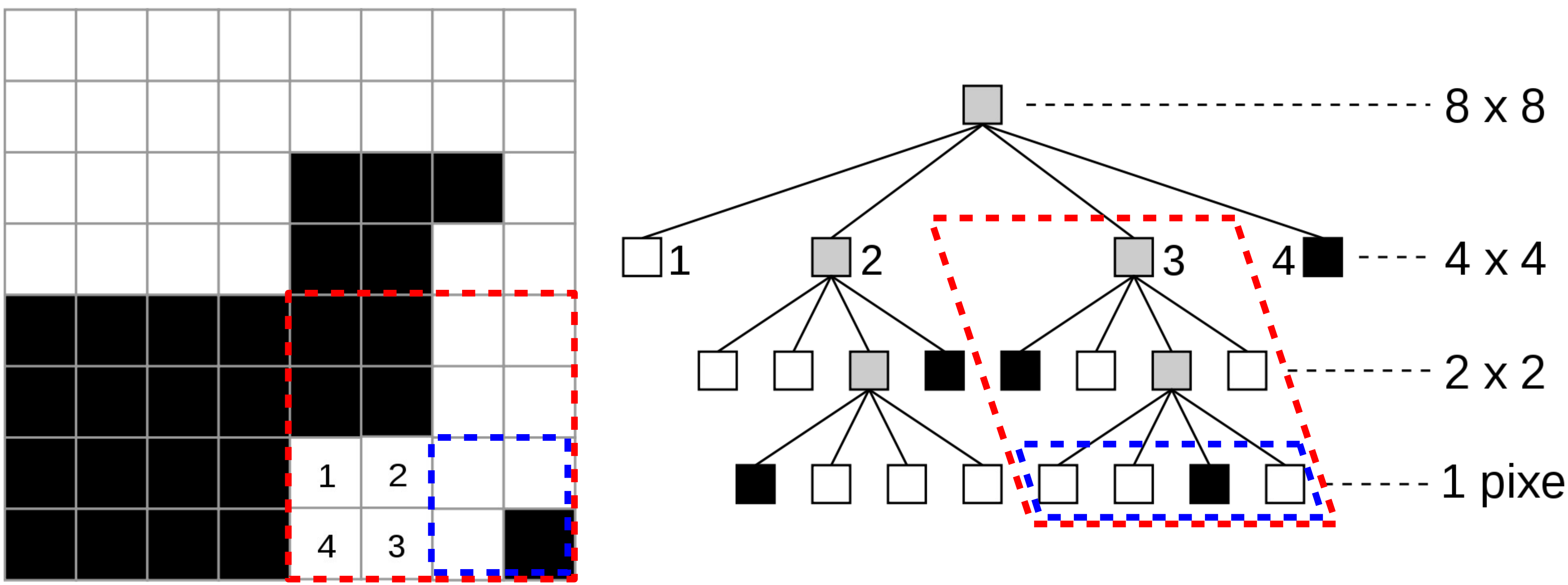
Géométrie de construction de solides (constructive solid geometry, CSG)



Sur un nœud non terminal :
une opération de composition
(union, intersection, différence...)

Sur une feuille :
un objet géométrique
élémentaire
(cube, sphère, cylindre...)

Exemple : représentation compacte d'une image Noir & Blanc



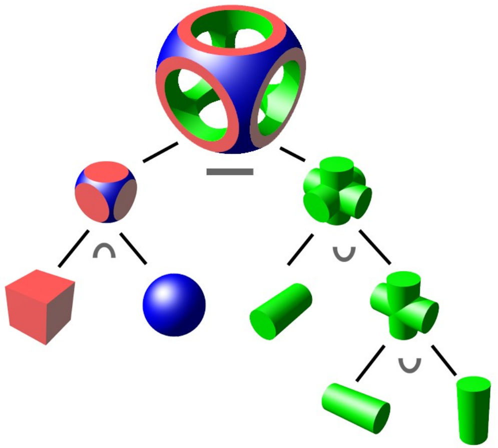
64 pixels

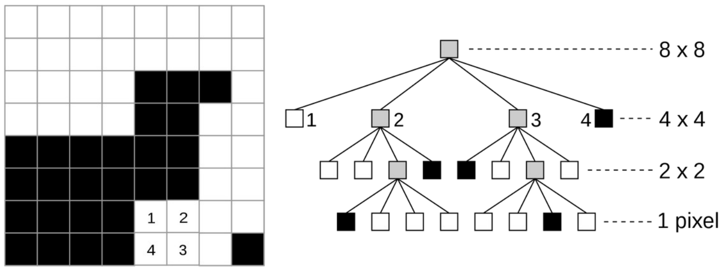
21 noeuds

info N&B : seulement sur les feuilles, pas sur les nœuds intermédiaires

Finalité vs moyen

- Arbre comme finalité
  - créé intentionnellement
  - opérations :
    - modifications : ajouter, retirer, déplacer...
    - parcours : recherche d'information, énumér
    - calcul : combinaisons des informations sur les nœuds...

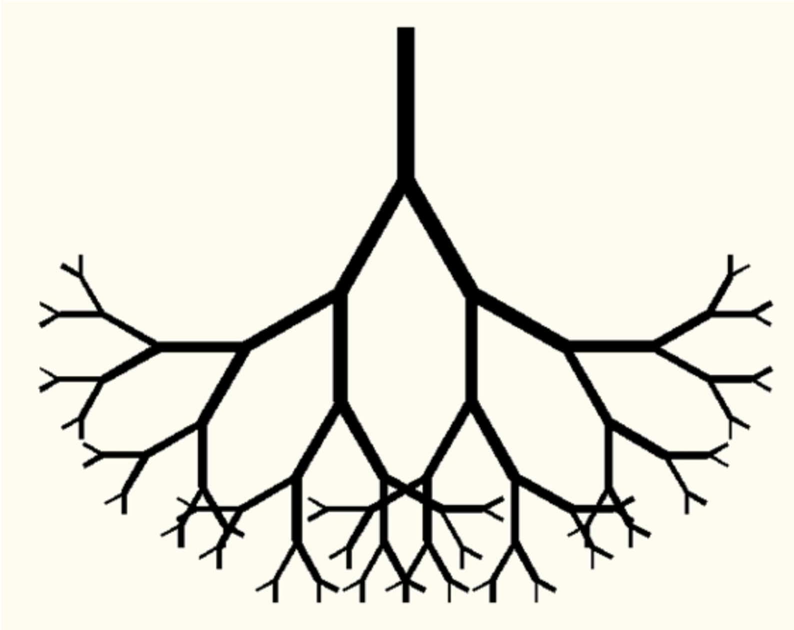




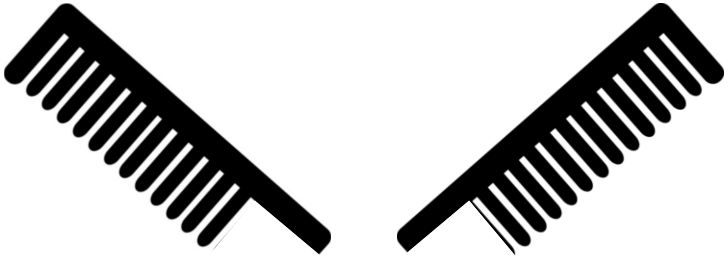
- Arbre comme moyen
  - support de données seulement
  - données modifiées ⇒ modification (automatique) de l'arbre

Arbre binaire

- Tout nœud a au plus deux enfants :
  - enfant gauche, enfant droit

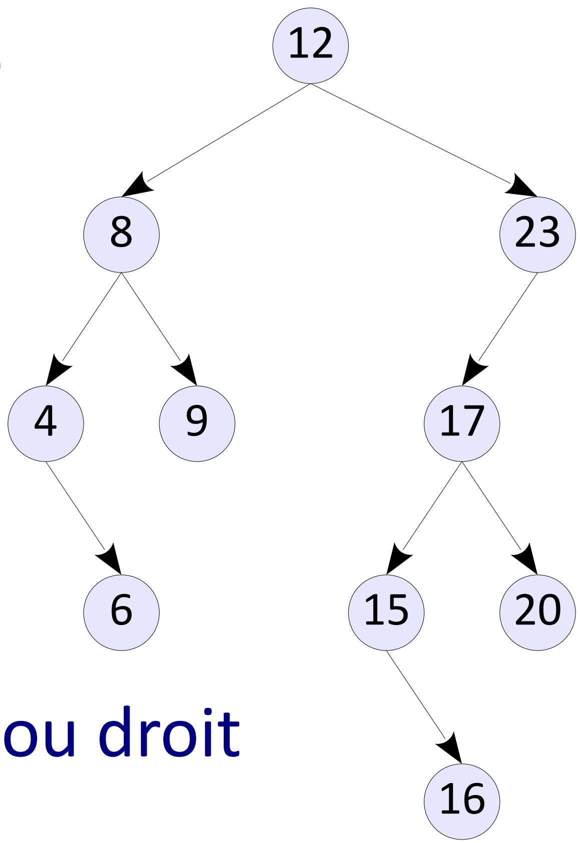


- Variante : pour tout nœud
  - 0 enfant (feuille), ou bien
  - exactement 2 enfants
- Cas dégénéré : peigne
  - (droit ou gauche)



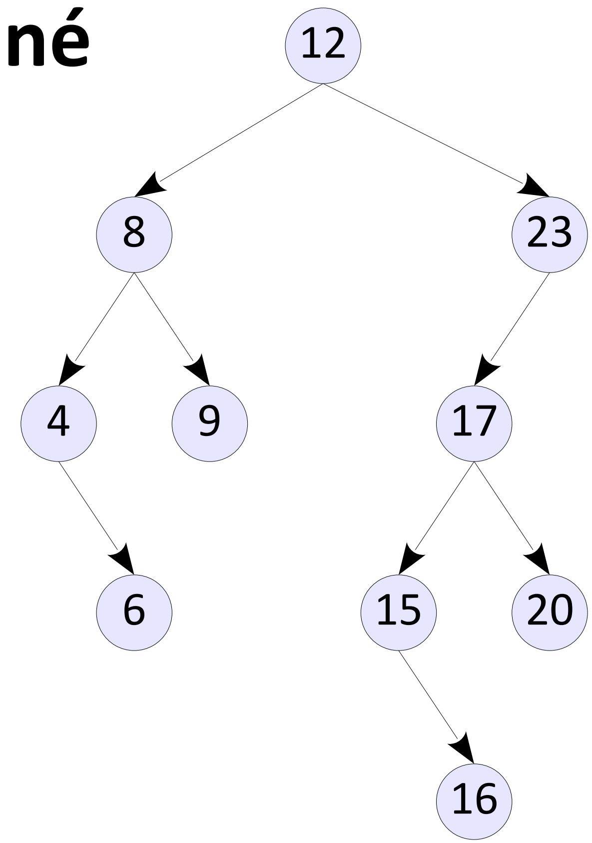
Arbre binaire de recherche

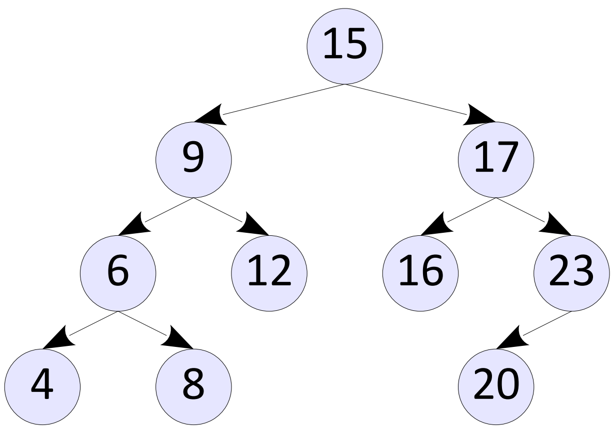
- Représentation d'un **ensemble ordonné**
  - ex. {4, 6, 8, 9, 12, 15, 16, 17, 20, 23}
  - info sur nœud nommée « clé »
- Propriété (pour tout nœud) :
  - clés sur sous-arbre gauche < clé du parent
  - clés sur sous-arbre droit > clé du parent
- Recherche ≈ par dichotomie
  - algo. : descente dans le sous-arbre gauche ou droit qui peut contenir la clé, ou arrêt si aucun
  - O(log *n*) en moyenne pour *n* nœuds [≠ liste, tableau : O(*n*)]
  - O(*n*) dans le pire cas (peigne)

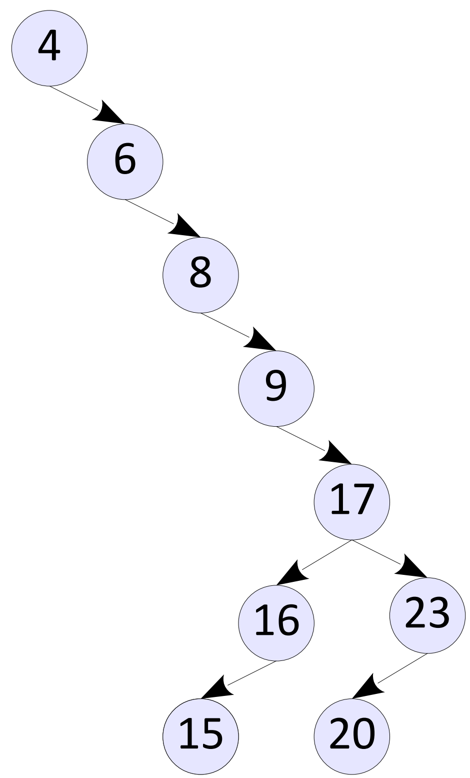


Arbre binaire de recherche

- Représentation d'un **ensemble ordonné**
  - ex. {4, 6, 8, 9, 12, 15, 16, 17, 20, 23}
  - arbre pas unique

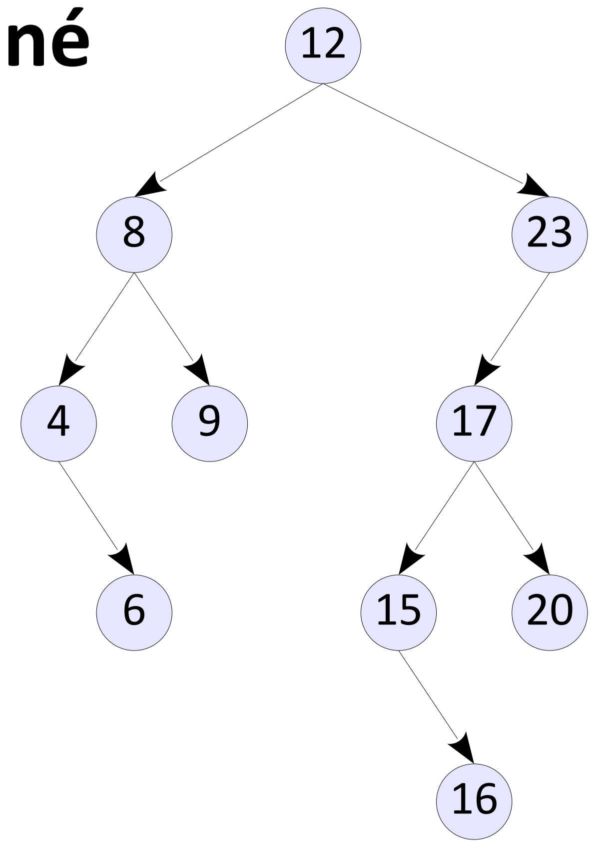


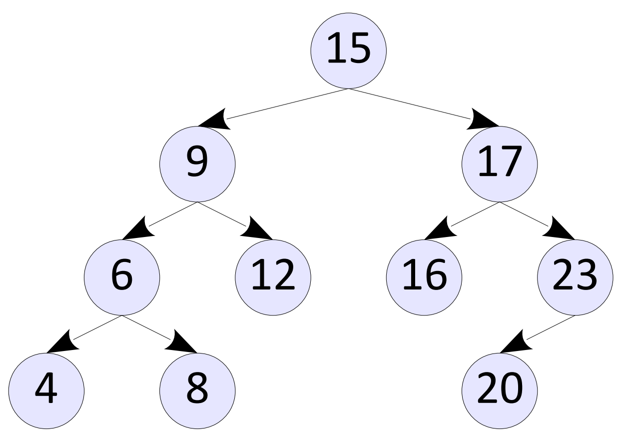


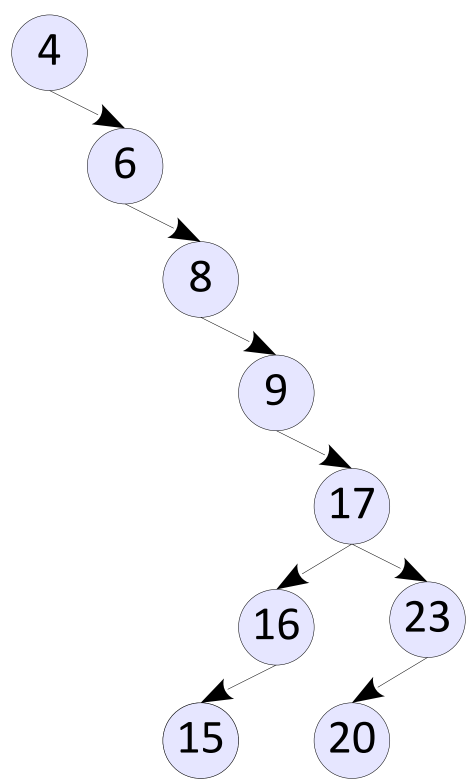


Arbre binaire de recherche

- Représentation d'un **ensemble ordonné**
  - ex. {4, 6, 8, 9, 12, 15, 16, 17, 20, 23}
  - arbre pas unique



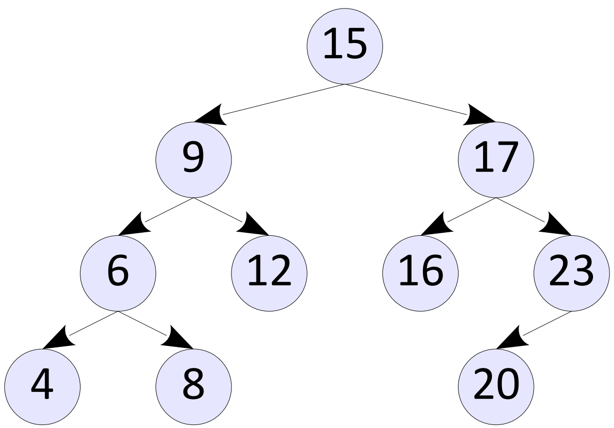




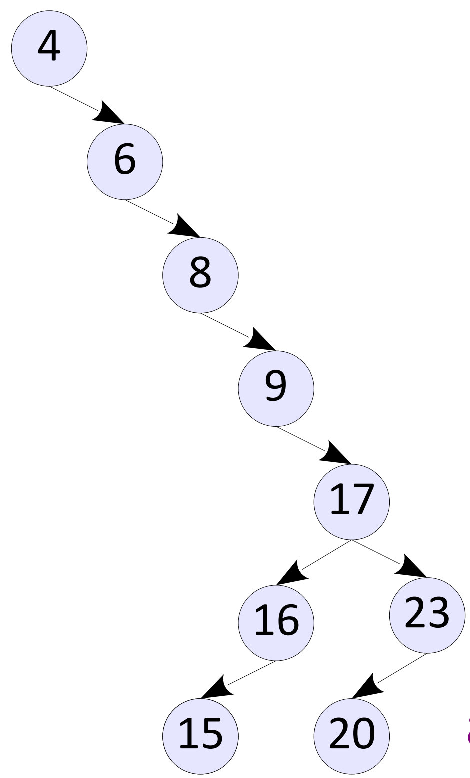
Laquelle est la meilleure ?

Arbre binaire de recherche

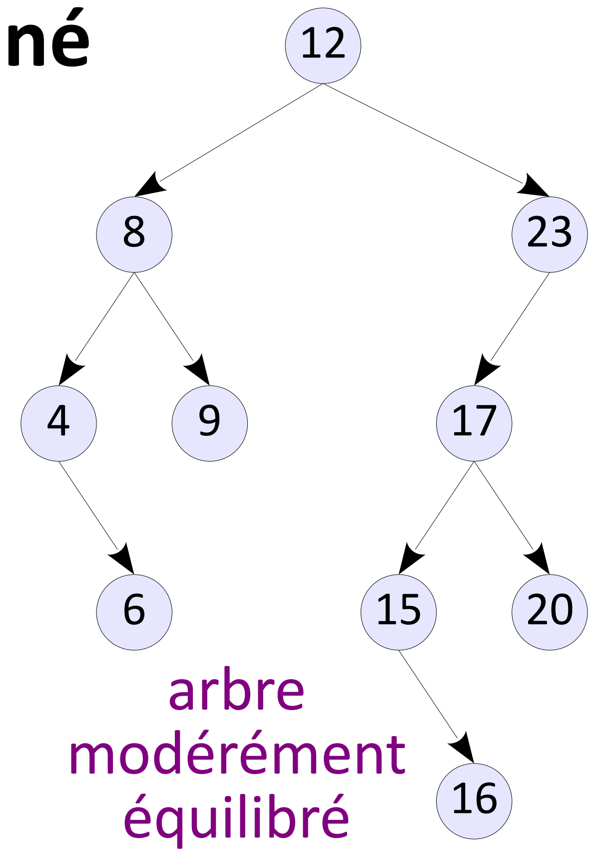
- Représentation d'un **ensemble ordonné**
  - ex. {4, 6, 8, 9, 12, 15, 16, 17, 20, 23}
  - arbre pas unique



arbre équilibré



arbre déséquilibré

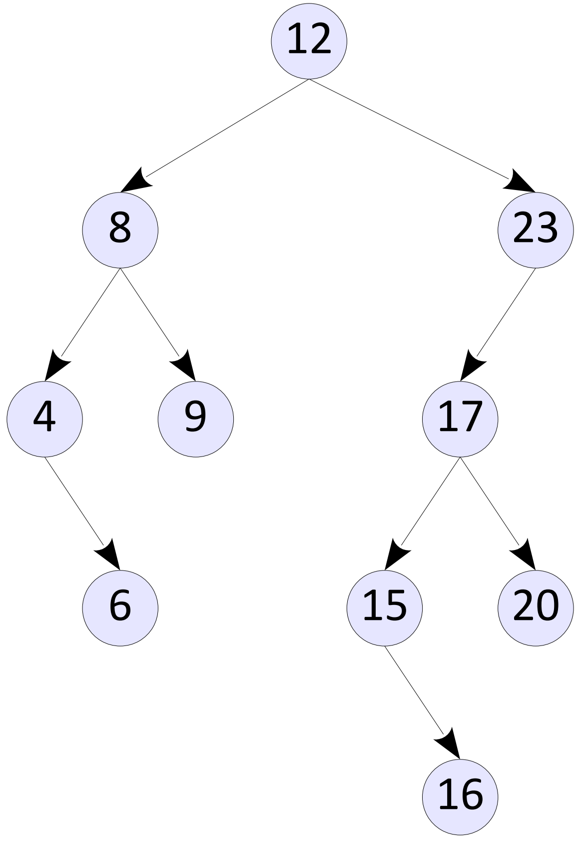


arbre modérément équilibré

Laquelle est la meilleure ?

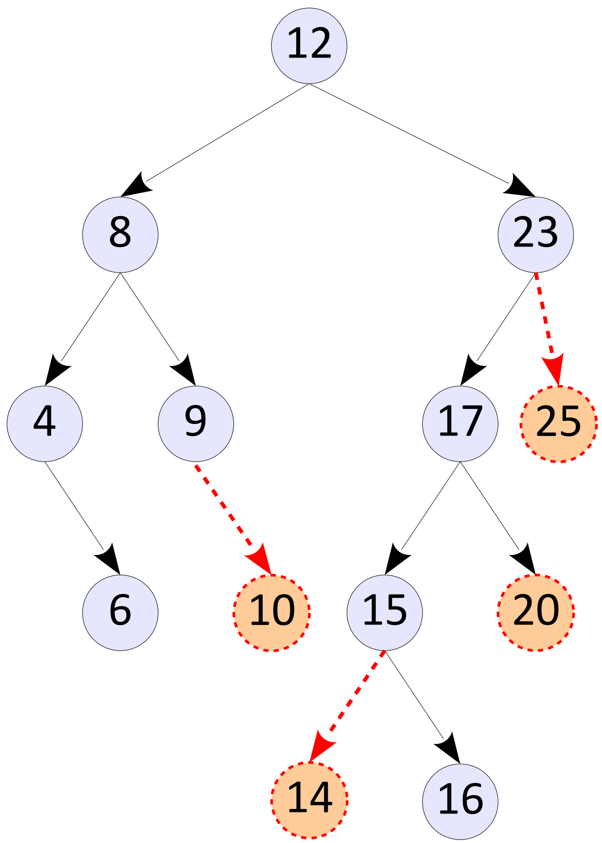
Insertion dans un arbre binaire de recherche

- Où peut-on (simplement) ...
  - insérer 10 ?
  - insérer 14 ?
  - insérer 20 ?
  - insérer 25 ?



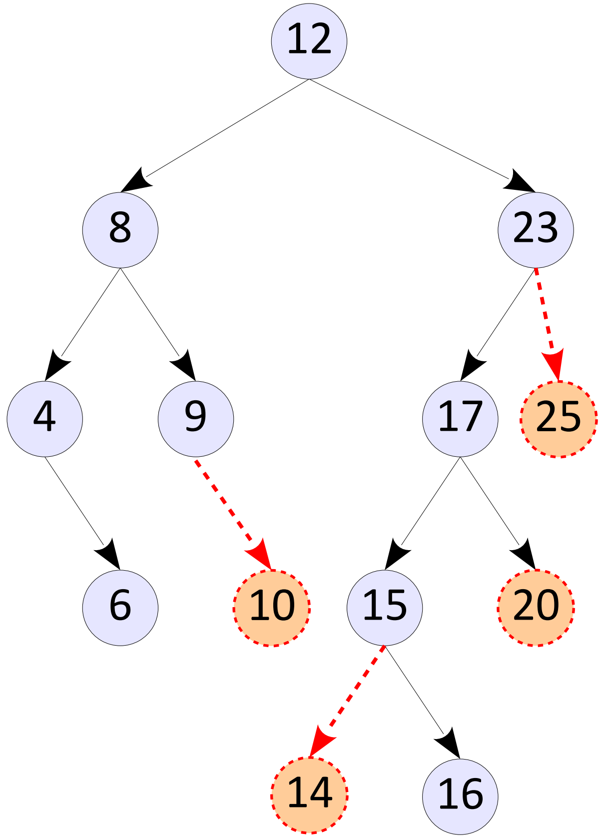
Insertion dans un arbre binaire de recherche

- Où peut-on (simplement) ...
  - insérer 10 ?
  - insérer 14 ?
  - insérer 20 ?
  - insérer 25 ?



Insertion dans un arbre binaire de recherche

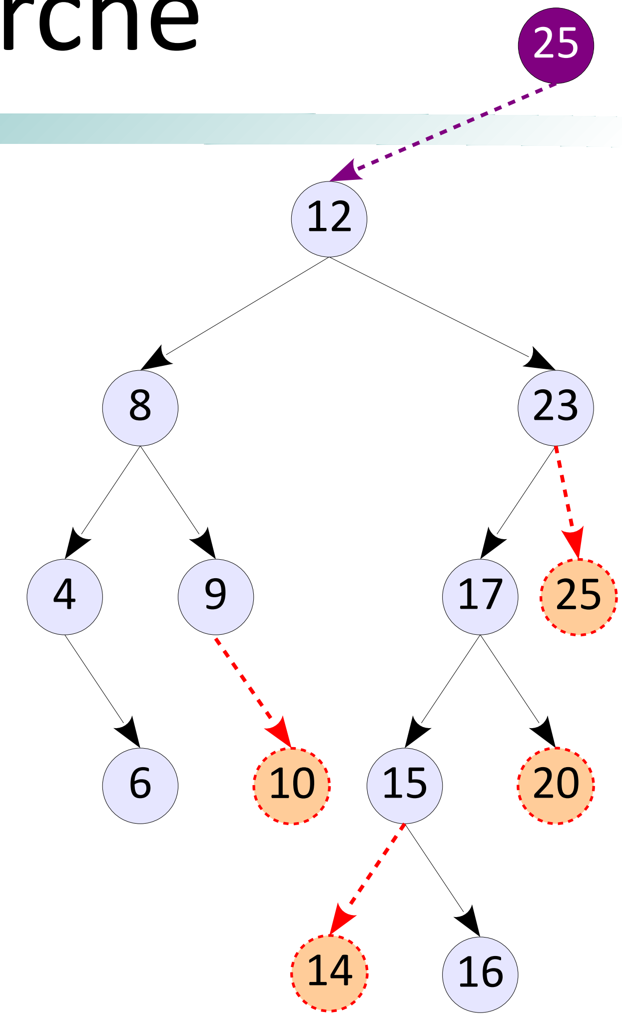
- Où peut-on (simplement) ...
  - insérer 10 ?
  - insérer 14 ?
  - insérer 20 ?
  - insérer 25 ?



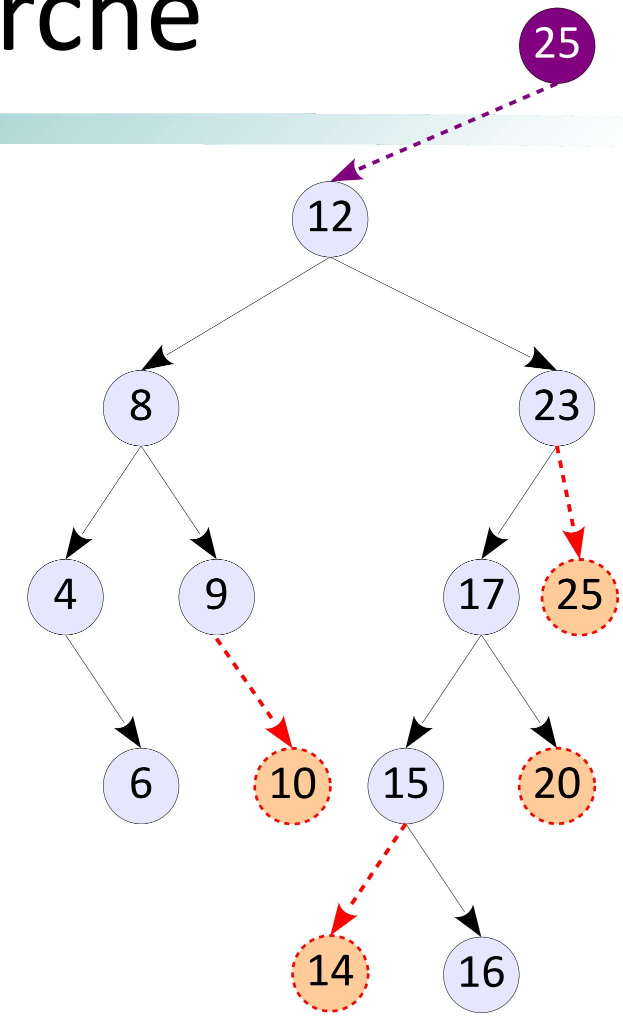
- Y a-t-il plusieurs endroits où insérer un nœud ?

Insertion dans un arbre binaire de recherche

- Où peut-on (simplement) ...
  - insérer 10 ?
  - insérer 14 ?
  - insérer 20 ?
  - insérer 25 ?
- Y a-t-il plusieurs endroits où insérer un nœud ?



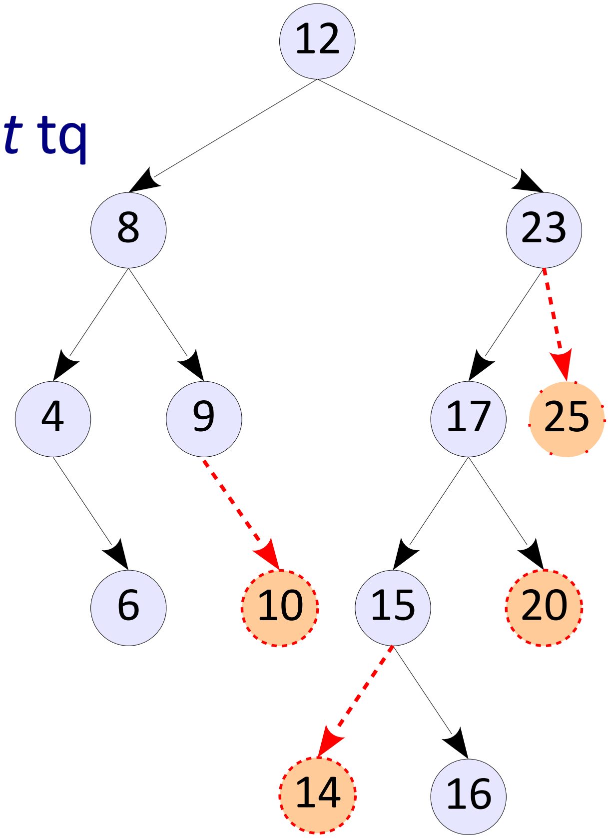
Insertion dans un arbre binaire de recherche



- Où peut-on (simplement) ...
  - insérer 10 ?
  - insérer 14 ?
  - insérer 20 ?
  - insérer 25 ?
- Y a-t-il plusieurs endroits où insérer le nœud ?
- Quel est l'algorithme ? Quelle est sa complexité ?

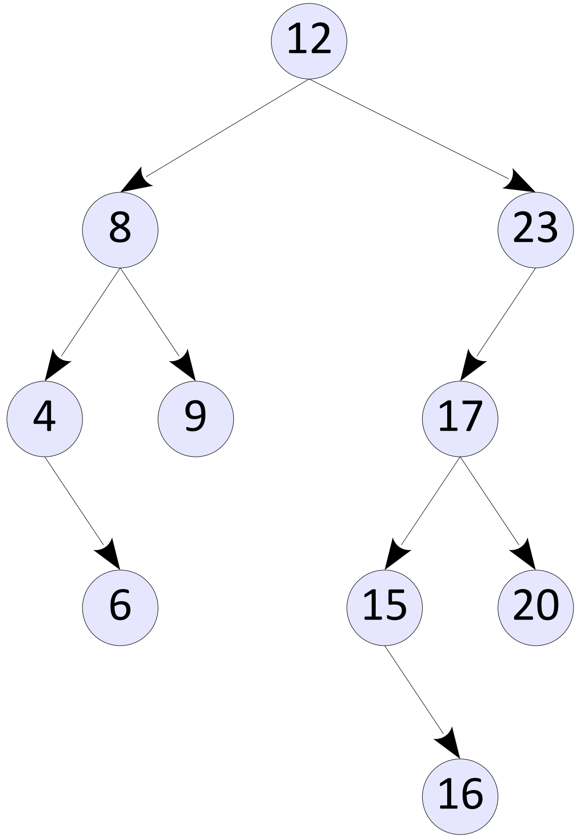
Insertion dans un arbre binaire de recherche

- Pour insérer la clé *x* :
  - descendre dans l'arbre en cherchant un nœud *t* tq
    - *x* < clé(*t*) et *t* n'a pas d'enfant gauche ou
    - *x* > clé(*t*) et *t* n'a pas d'enfant droit ou
  - si *x* = clé(*t*), s'arrêter [*x* déjà présent]
  - si *x* ≠ clé(*t*), créer un nouvel enfant de *t*, resp. gauche ou droit, avec un nouveau noeud *x*
- Complexité
  - comme pour la recherche
    - O(log(*n*)) en moyenne, si arbre à peu près équilibré
    - O(*n*) dans le pire cas



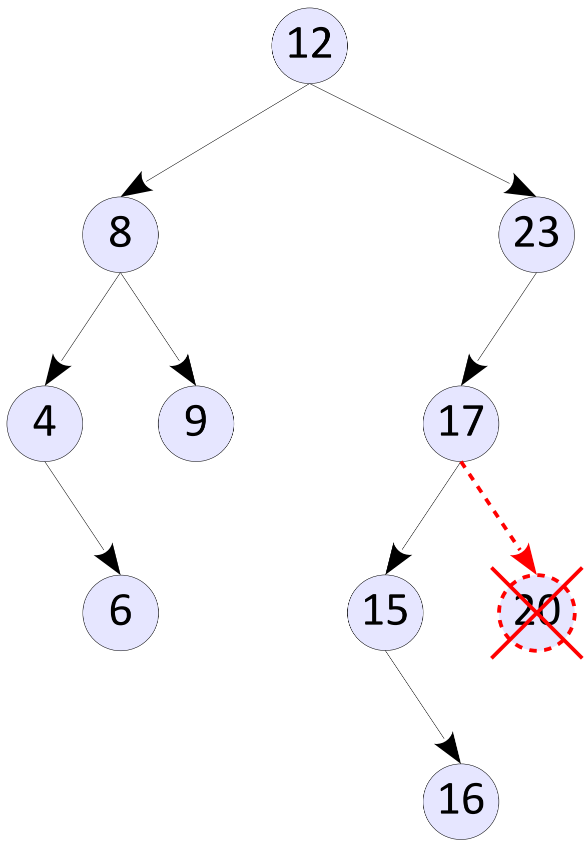
Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...
  - supprimer 20 ?



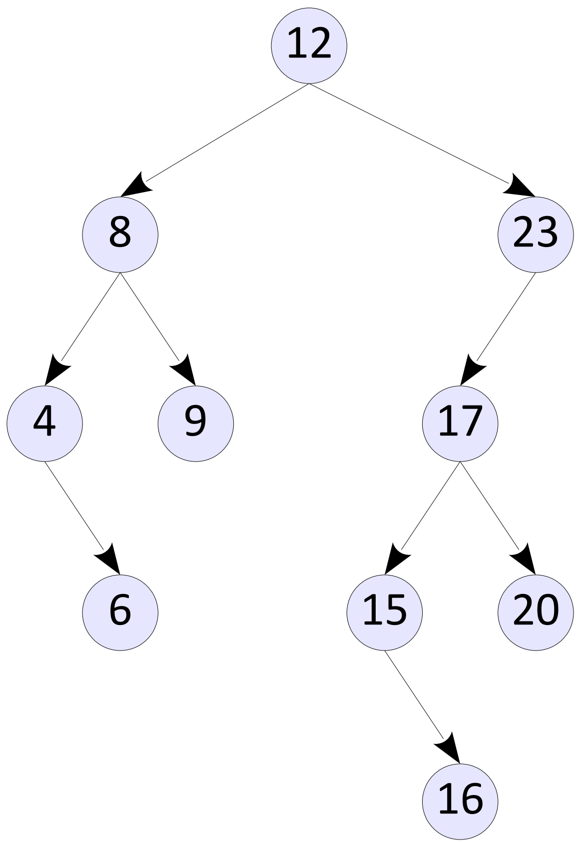
Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...
  - supprimer 20 ?



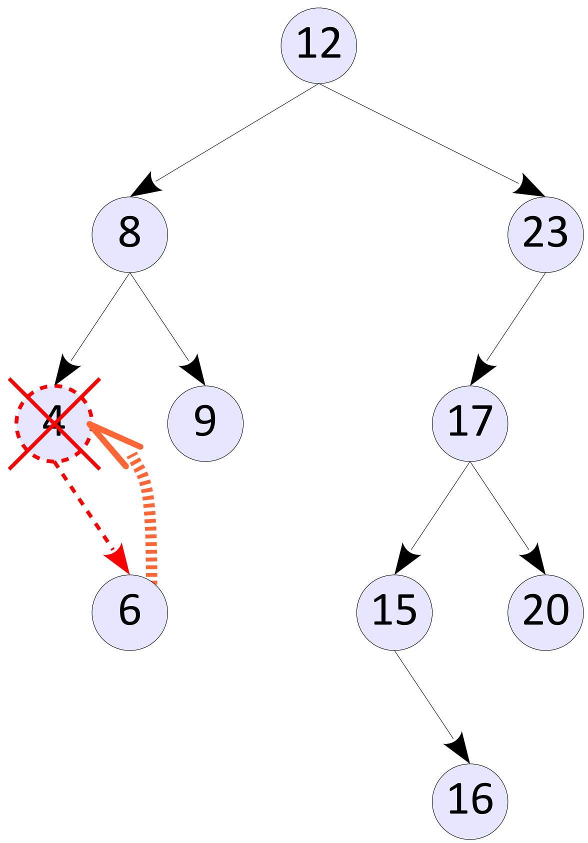
Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...
  - supprimer 4 ?



Suppression dans un arbre binaire de recherche

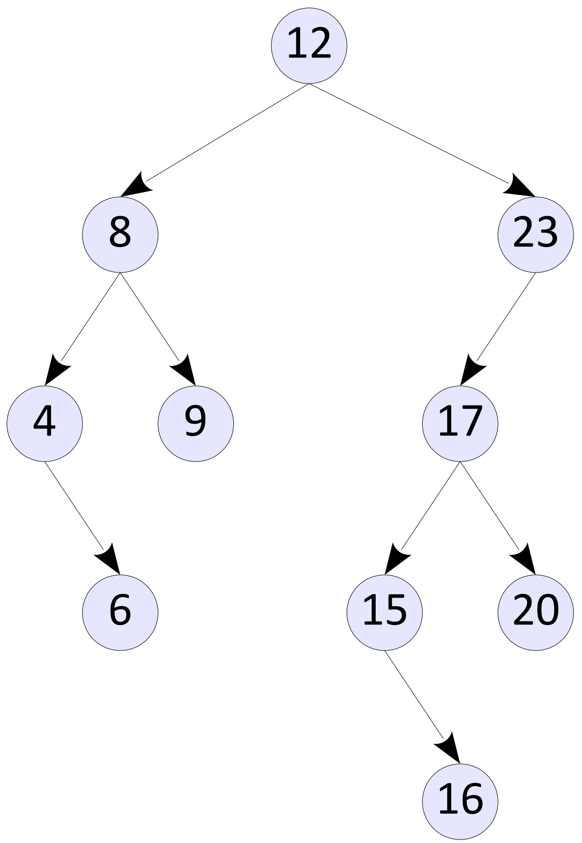
- Comment peut-on (simplement) ...
  - supprimer 4 ?



Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...

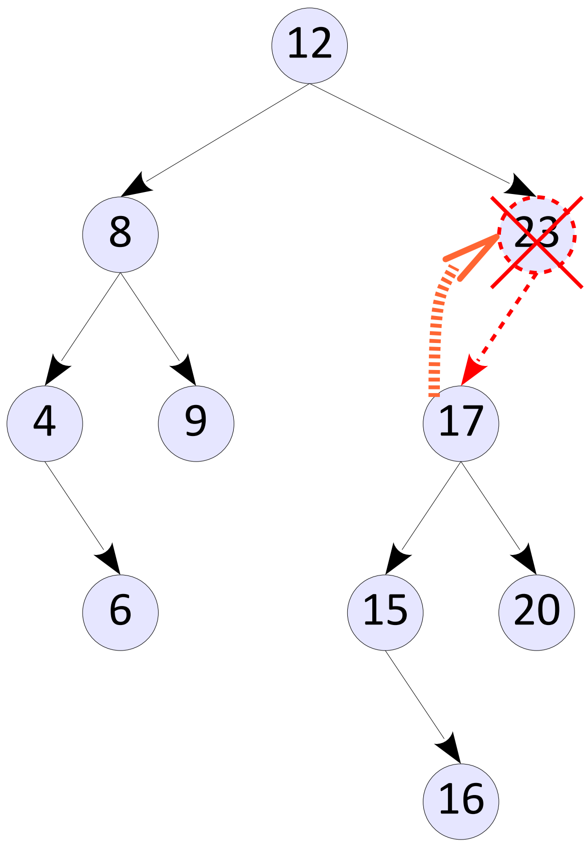
  - supprimer 23 ?



Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...

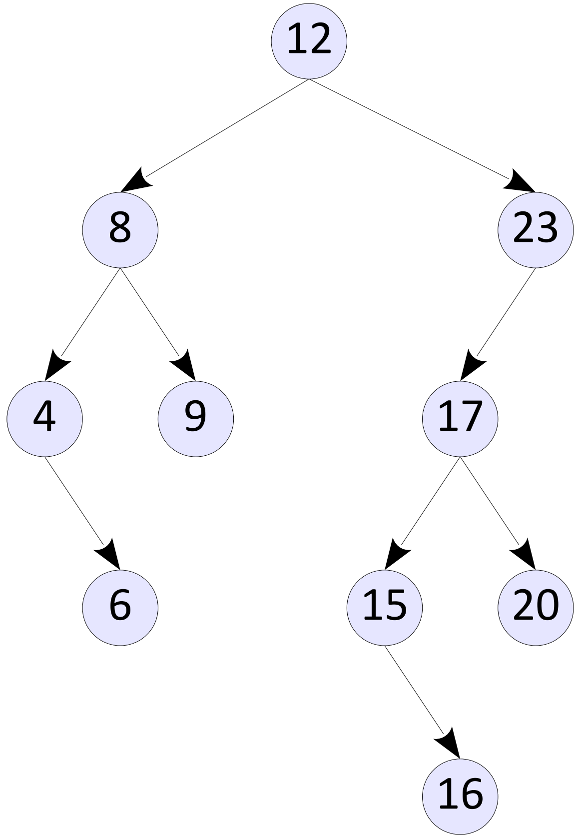
  - supprimer 23 ?



Suppression dans un arbre binaire de recherche

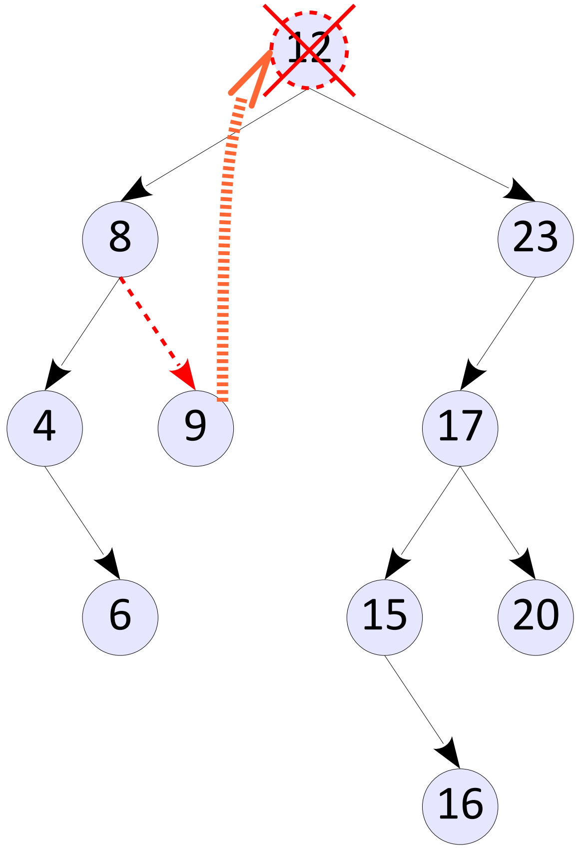
- Comment peut-on (simplement) ...

  - supprimer 12 ?



Suppression dans un arbre binaire de recherche

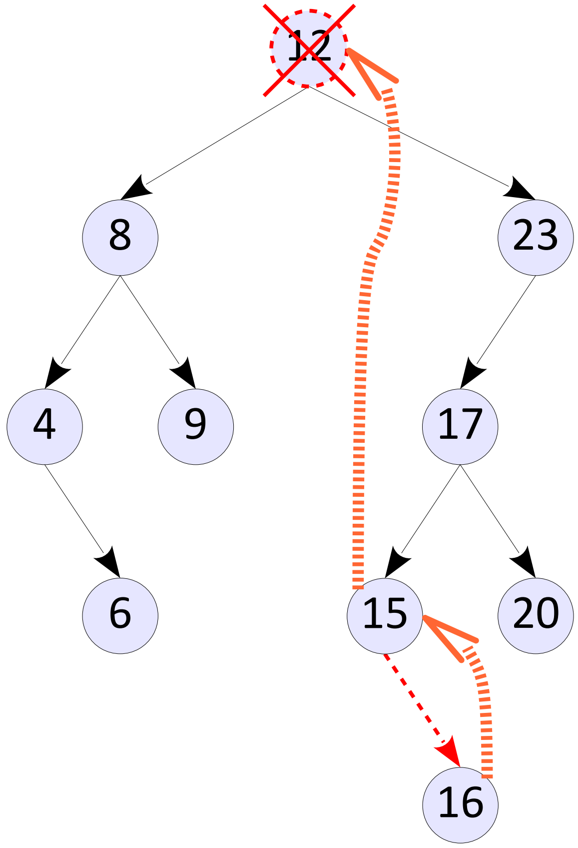
- Comment peut-on (simplement) ...
  - supprimer 12 ?



Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...

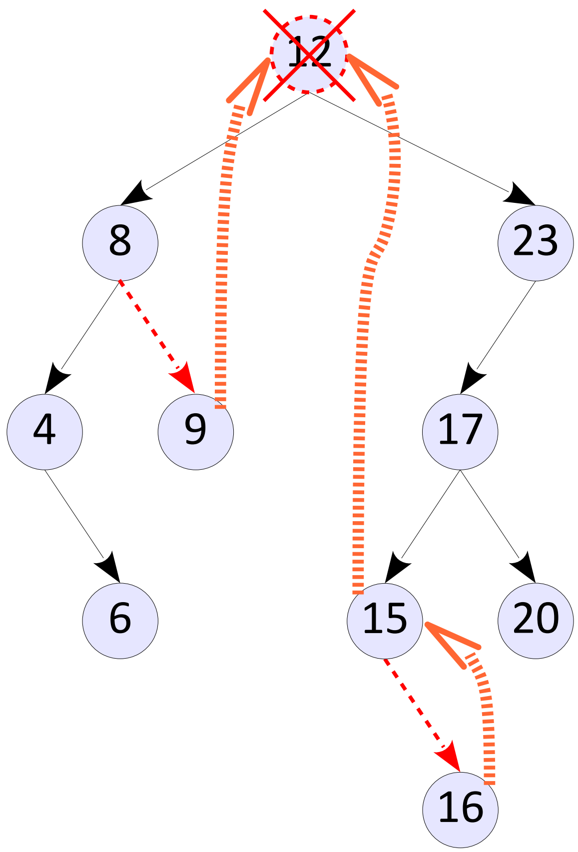
  - supprimer 12 ?



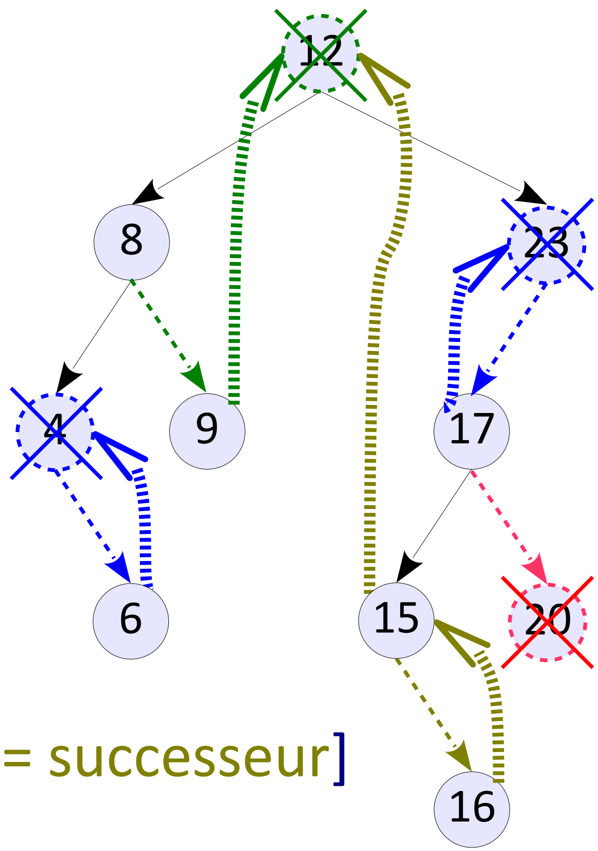
Suppression dans un arbre binaire de recherche

- Comment peut-on (simplement) ...
  - supprimer 20 ?
  - supprimer 4 ?
  - supprimer 23 ?
  - supprimer 12 ?

- Quel est l'algorithme ?
  Quelle est sa complexité ?



Suppression dans un arbre binaire de recherche



- Pour supprimer la clé *x* :
  - rechercher le nœud *t* avec la clé *x*
  - si *t* est une feuille (ex. suppr. 20),
    - la supprimer
  - si *t* a un enfant (ex. suppr. 4, 23),
    - supprimer *t* et le remplacer par son fils
  - si *t* a deux enfants (ex. suppr. 12),
    - chercher *t'* le nœud le plus à droite du sous-arbre gauche (= le prédécesseur de *x*) [ou symétrique : plus à gauche de la droite = successeur]
    - remplacer clé(*t*) par clé(*t'*)
    - supprimer *t'* (qui n'a que 0 ou 1 fils), c.-à-d. attribuer comme enfant du parent de *t'* l'éventuel fils gauche [resp. droite] de *t'*
- Complexité : O(log(*n*)) en moyenne si équilibré, O(*n*) dans pire cas

Exemple d'interface

```
class BinarySearchTree { // En fait, un nœud...
    int key; // Ici juste un entier pour simplifier
    BinarySearchTree *left, *right; // nullptr ⇔ pas de fils
public:
    BinarySearchTree(int k);// Crée un arbre de clé k (nœud sans fils)
    ~BinarySearchTree();
    bool contains(int k) const;// Teste si l'arbre contient k
    void insert(int k);        // Insert k dans l'arbre (si absent)
    void remove(int k);        // Supprime k de l'arbre (si présent)
};
```

Pointeur nul en C++11

Exercice d'entraînement 1 (1/3)

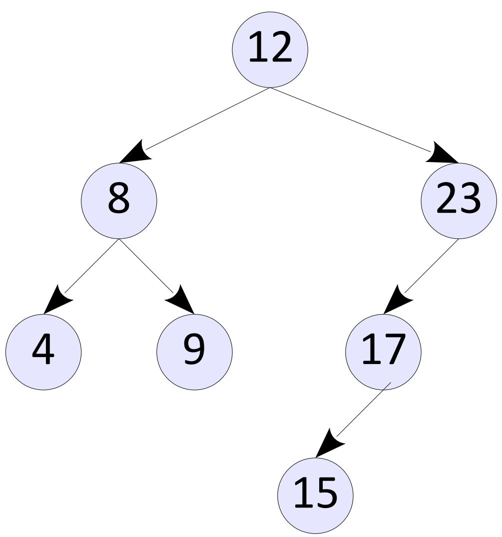
1.1) Rendre publics les champs **`left`** et **`right`** <u>pour le debug</u>

1.2) Construire « à la main » l'arbre ci-dessous
dans une variable **`BinarySearchTree* bst;`**

1.3) Ajouter une fonction récursive
`void display(string prefix = "")`
telle que **`bst.display()`** affiche :

```
  / 23
  /   \ 17
  /   \ \ 15
12
  \   / 9
  \ 8
  \   \ 4
```

à lire en penchant la tête
sur le côté gauche…



Exercice d’entraînement 1 (2/3)

1.4) Définir et tester la fonction **`contains`**

1.5) Définir et tester la fonction **`insert`**

- créer un arbre initial avec la valeur 12,
  puis y insérer les valeurs 8, 4, 9, 23, 17, 15, et l'afficher
- faire des essais avec des ordres d'insertion différents
- dans quel cas général l'arbre résultant est-il déséquilibré ?

1.5) Définir et tester la fonction **`remove`**

1.6) Après débogage, rendre à nouveau privés les champs **`left`** et **`right`**: seuls **`insert`**/**`remove`** peuvent maintenant les modifier

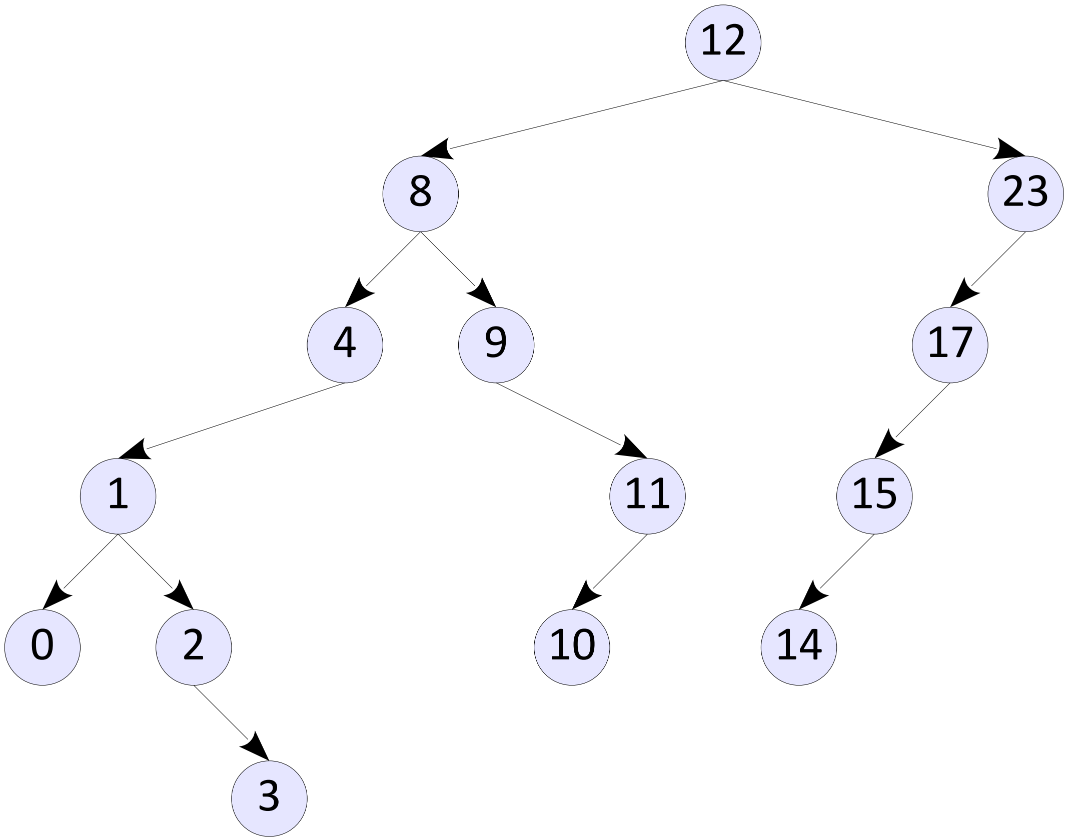
1.7) Faire un programme qui illustre successivement différents cas d'insertion, de tests d'appartenance et de suppression

Exercice d’entraînement 1 (3/3)

1.8) Modifier la fonction **`display()`** (< 10 lignes) pour qu'elle affiche les arbres comme ceci :

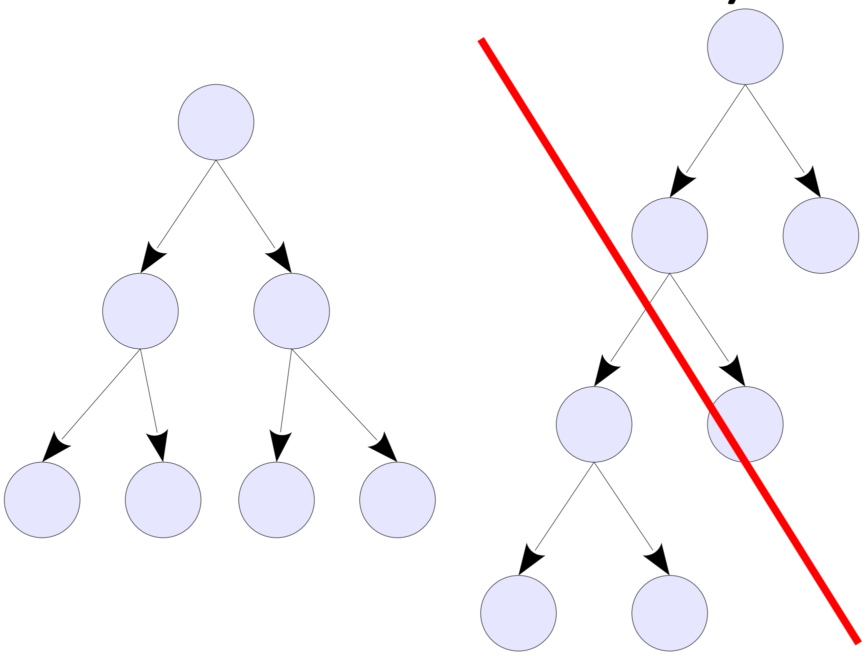
à lire sur le côté…

```
/--23
|  \--17
|     \--15
|        \--14
12
|     /--11
|     |  \--10
|  /--9
\--8
   \--4
      |     /--3
      |  /--2
      \--1
         \--0
```



Arbre équilibré

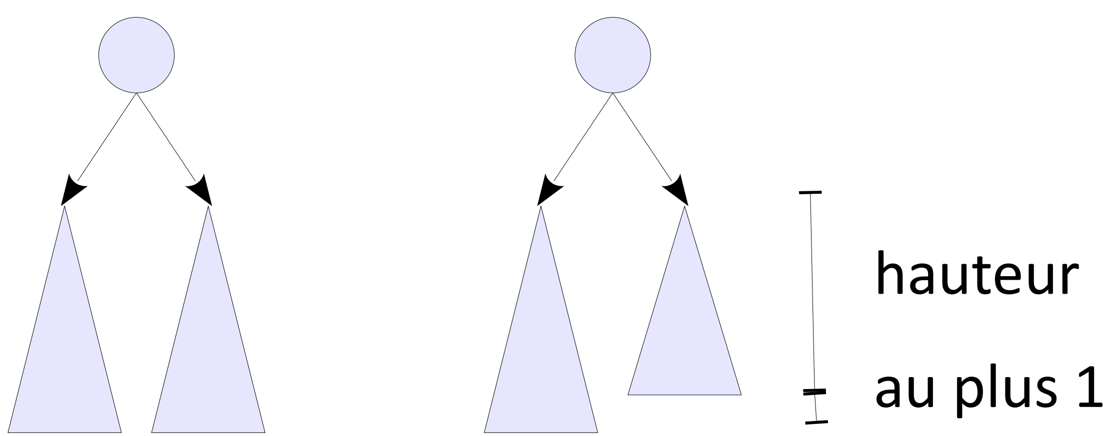
- Arbre avec critère d'équilibre (ex. nb nœuds des fils)
- Arbre binaire équilibré :
  - critère d'équilibre
    - entre fils gauche et fils droit
    - total ou partiel
  - évite notamment les peignes



- Avantages
  - recherche en O(log(n)) en **pire cas** pour un arbre à n nœuds
  - insertion et suppression O(n) ou O(log(n)) suivant variantes

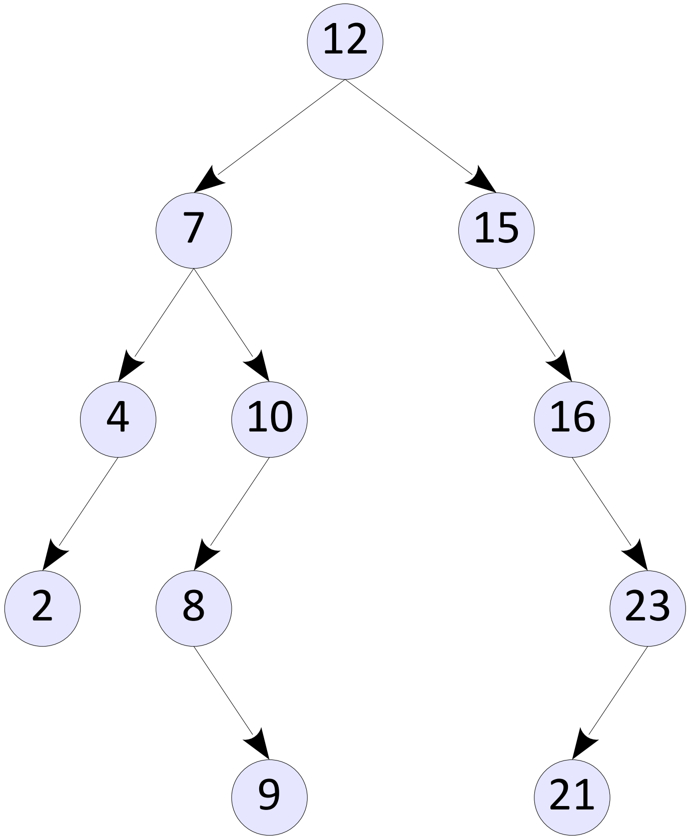
Ex. Arbre AVL

- Georgii Adelson-Velsky & Evguenii Landis (1962)
  - principe général toujours d'actualité
- Arbre AVL
  - arbre binaire de recherche équilibré tq la hauteur de deux sous-arbres d'un même nœud diffère au plus de 1



Arbre non AVL : exemple

arbre non AVL

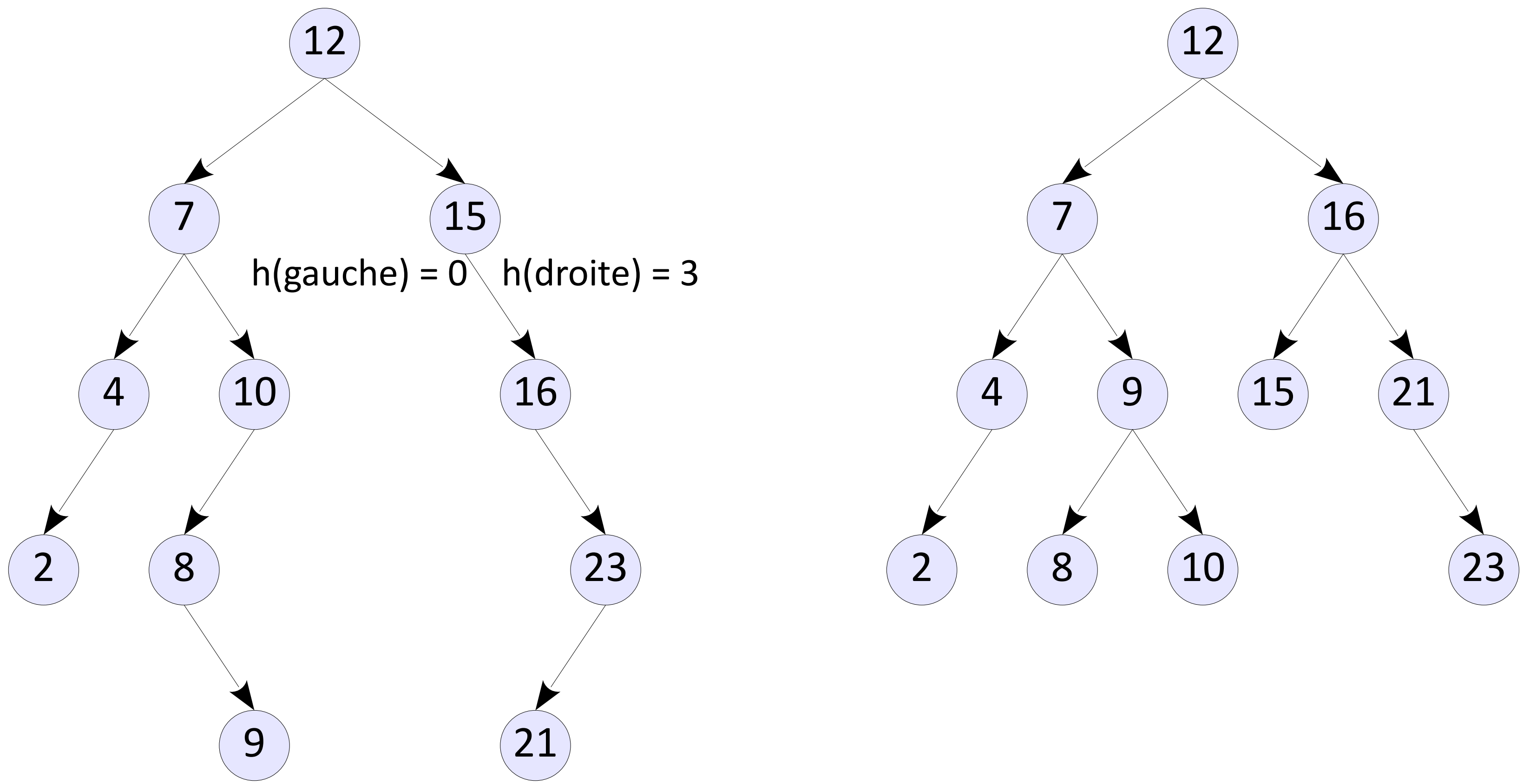


Pourquoi ?

Arbre AVL : exemple

arbre non AVL

arbre AVL (après rééquilibrage)



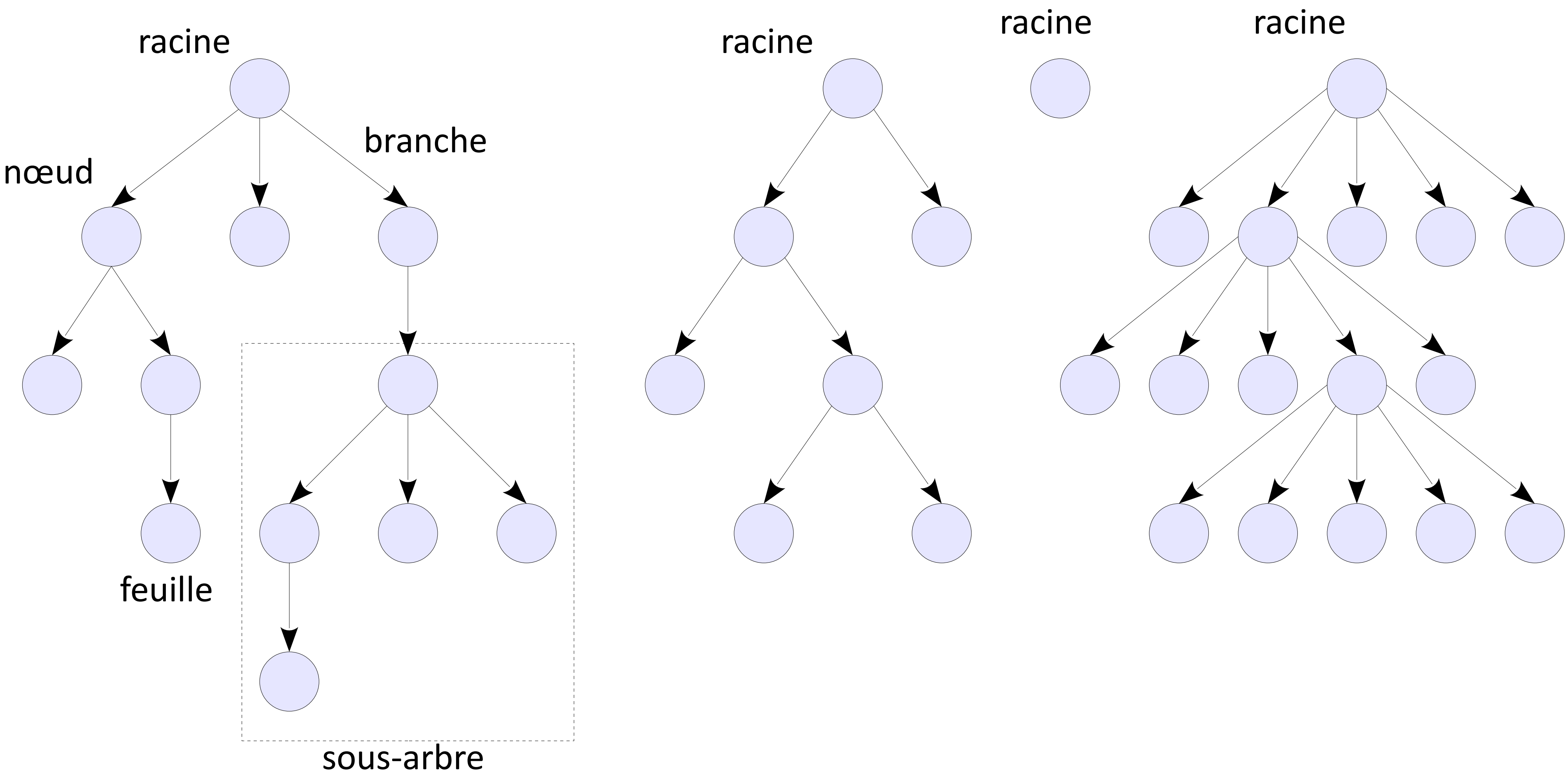
Arbre AVL

- Propriété
  - recherche, insertion, suppression en O(log(n))
- Principe (**transposable dans d'autres circonstances**)
  - insertion et suppression font du rééquilibrage « au vol »
- Voir : plein d'exemples sur le web...

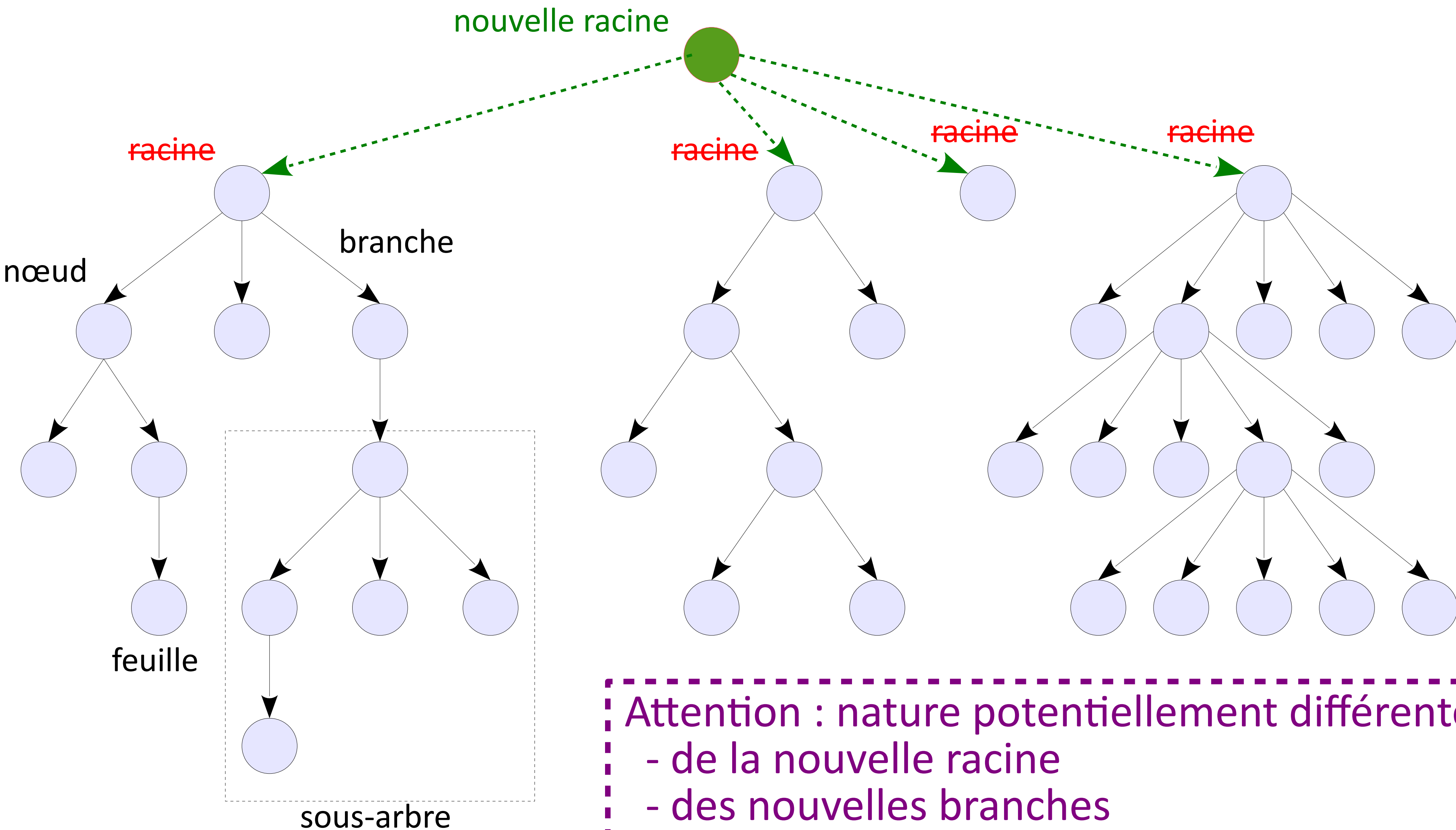
  (par ex. http://oopweb.com/Algorithms/Documents/AvlTrees/Volume/AvlTrees.htm)

Forêt

- Forêt = ensemble d'arbres (!...)



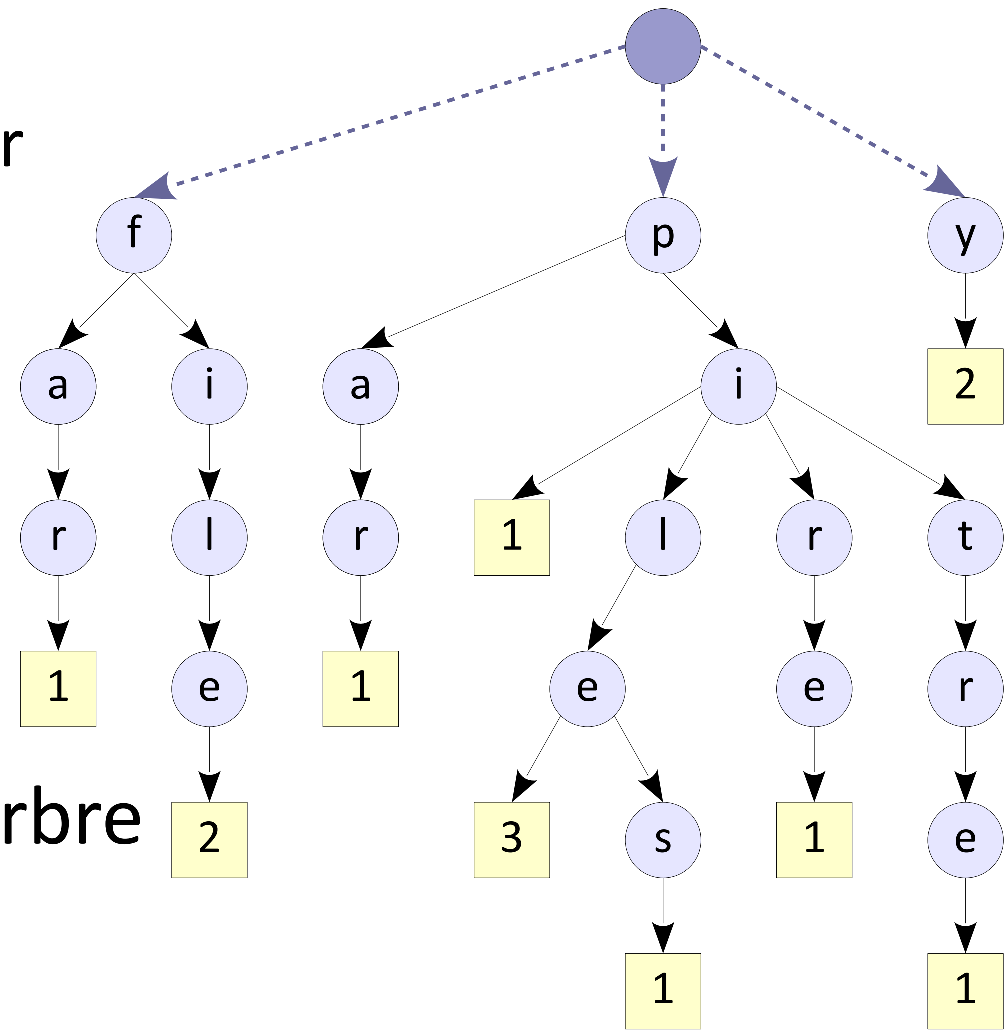
Forêt rassemblée en un « super-arbre »



Attention : nature potentiellement différente
- de la nouvelle racine
- des nouvelles branches

Exemple : « arbre » préfixe (trie)

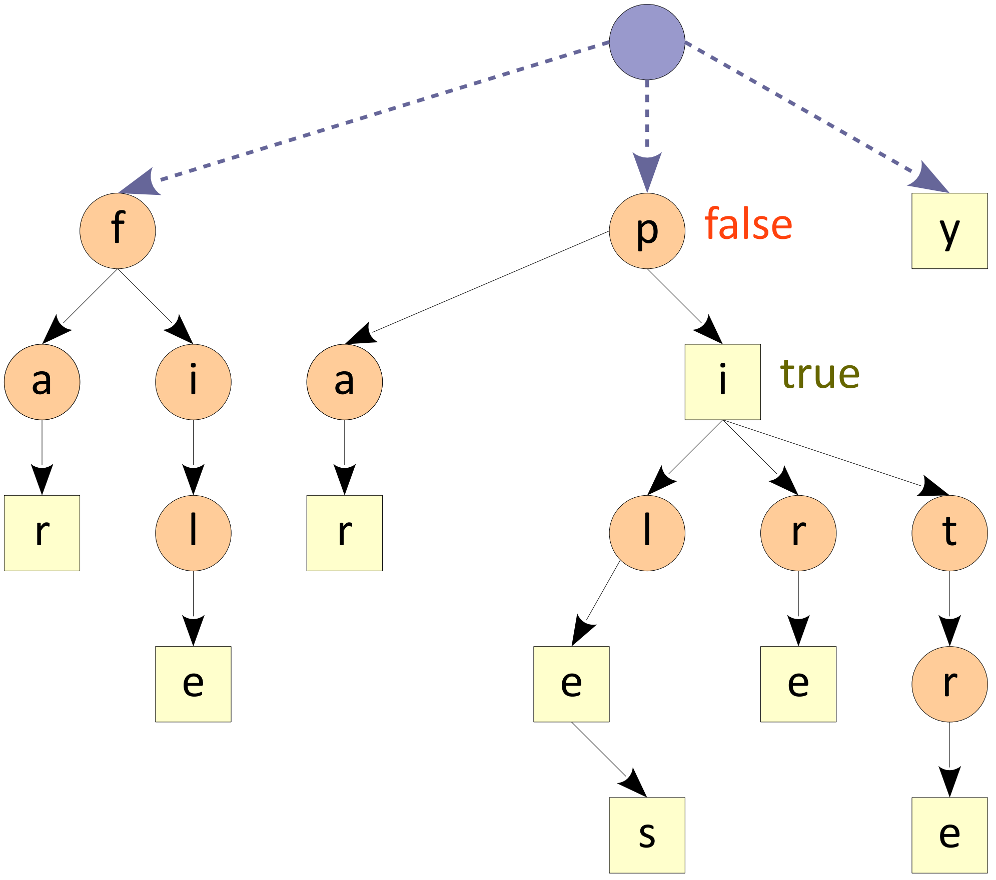
- Table associative
  - chaîne de char. → valeur
- Ex. nb d'occurrences
  - Texte :
    far y file pile
    pire pitre
    pile file piles
    pi pile y par



- En fait, plus forêt que arbre
- De l'anglais re*trie*val

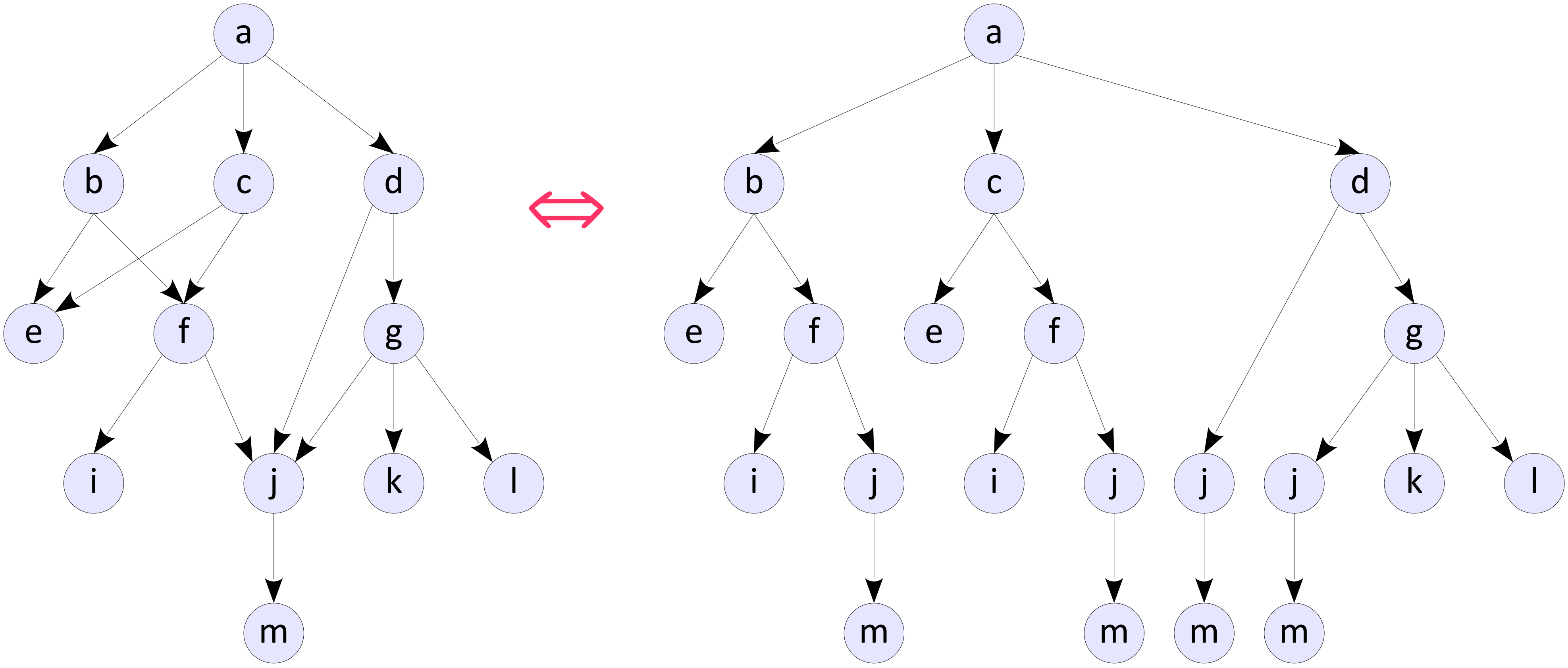
Arbre préfixe : dictionnaire de séquences

- Lexique
  - chaîne → {true, false}
- Ex. occurrence
  - Texte :
    far y file pile
    pire pitre
    pile file piles
    pi pile y par
- Également utilisé :
  - arbre des suffixes



Arbre avec partage des nœuds

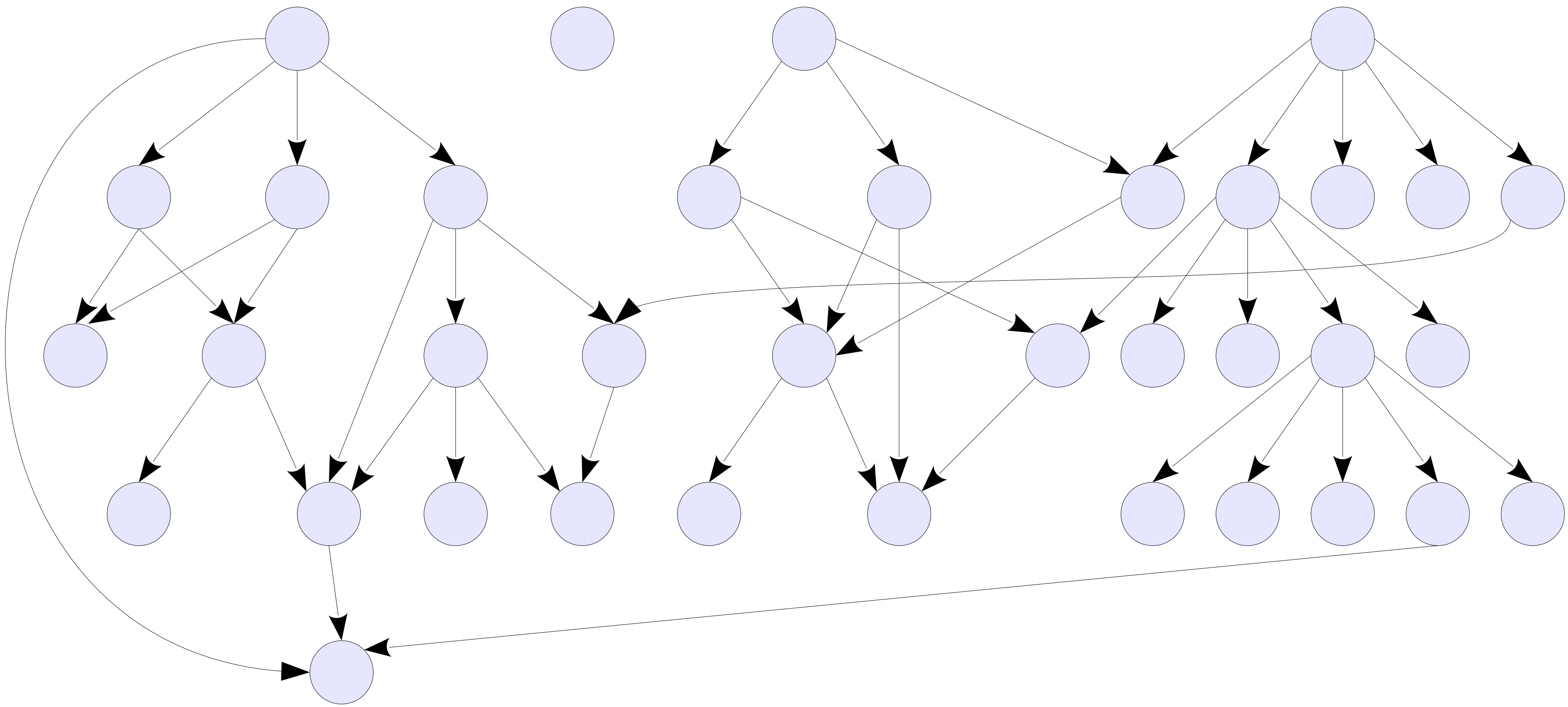
- Jusqu'à exponentiellement plus compact



Quel exemple de configuration procure un gain exponentiel ?

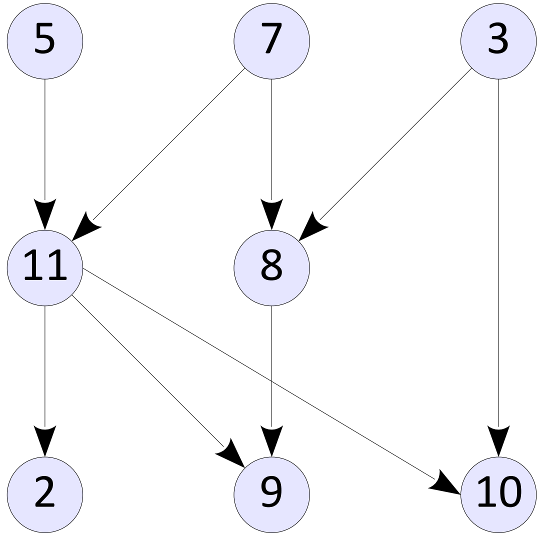
Graphe orienté acyclique

- DAG (directed acyclic graph) = arbre/forêt partagé(e)
  - jusqu'à exponentiellement plus compact



Tri topologique (1)

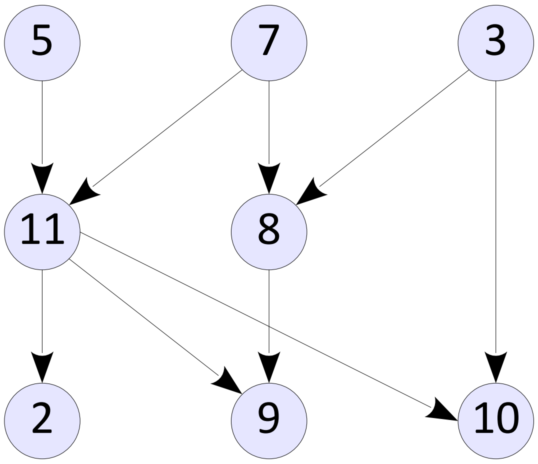
- Tri topologique d'un DAG
  - fournit un ordre linéaire « compatible » avec le DAG
  - c.-à-d. un ordre de visite des sommets tel qu'un sommet est toujours visité avant ses successeurs
- Exemple : 7, 5, 3, 11, 8, 2, 9, 10
- Applications
  - ordonnancement de tâches
    - gestion de projet, P.E.R.T.
      (Program Evaluation and Review Technique)
  - recompilation (make)



Tri topologique (2)

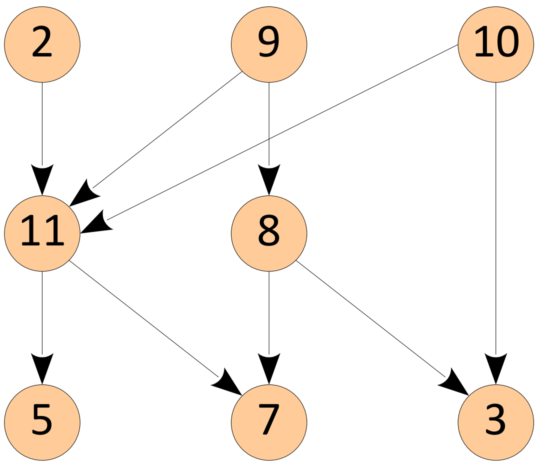
- Pas d'unicité
  - 7, 5, 3, 11, 8, 2, 9, 10
  - 3, 5, 7, 8, 11, 2, 9, 10
  - 3, 7, 8, 5, 11, 10, 2, 9
  - 5, 7, 3, 8, 11, 10, 9, 2
  - 7, 5, 11, 3, 10, 8, 9, 2
  - 7, 5, 11, 2, 3, 8, 9, 10
  - …
- Tri topologique inverse

  = pour la relation inverse (flèches inversées)
  - 10, 9, 8, 3, 2, 11, 7, 5
  - …



DAG

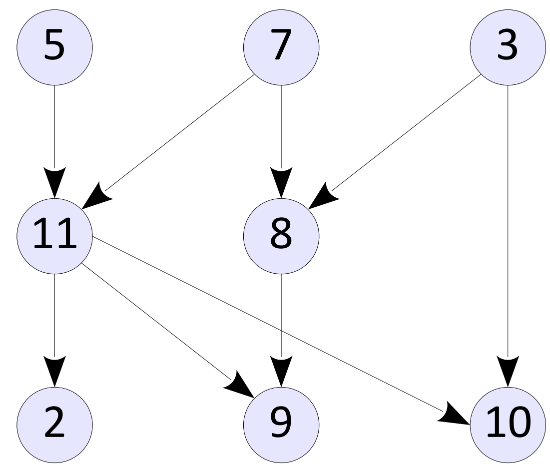
image miroir



DAG de la relation inverse

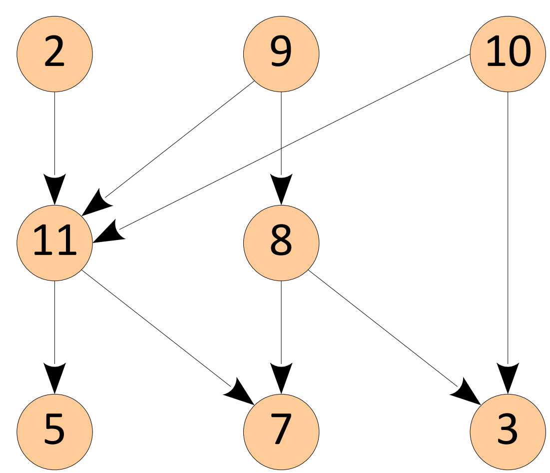
Tri topologique (2)

- Pas d'unicité
  - 7, 5, 3, 11, 8, 2, 9, 10
  - 3, 5, 7, 8, 11, 2, 9, 10
  - 3, 7, 8, 5, 11, 10, 2, 9
  - 5, 7, 3, 8, 11, 10, 9, 2
  - 7, 5, 11, 3, 10, 8, 9, 2
  - 7, 5, 11, 2, 3, 8, 9, 10
  - ...
- Tri topologique inverse
  = pour la relation inverse (flèches inversées)
  - 10, 9, 8, 3, 2, 11, 7, 5
  - ...



DAG

image miroir



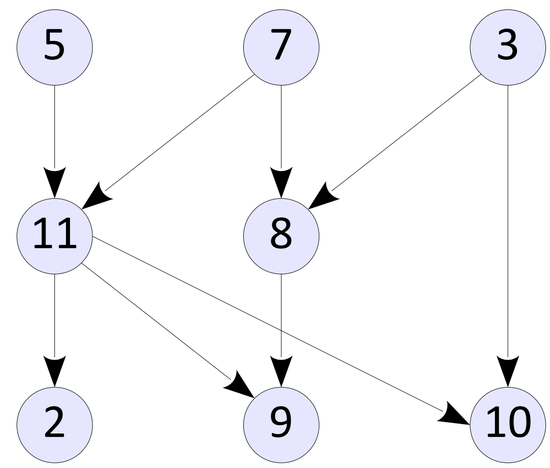
Pourquoi est-ce aussi un DAG ?

DAG de la relation inverse

Tri topologique (2)

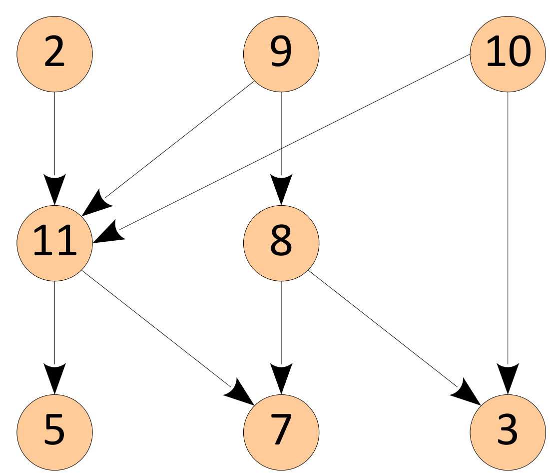
- Pas d'unicité
  - 7, 5, 3, 11, 8, 2, 9, 10
  - 3, 5, 7, 8, 11, 2, 9, 10
  - 3, 7, 8, 5, 11, 10, 2, 9
  - 5, 7, 3, 8, 11, 10, 9, 2
  - 7, 5, 11, 3, 10, 8, 9, 2
  - 7, 5, 11, 2, 3, 8, 9, 10
  - ...
- Tri topologique inverse

  = pour la relation inverse (flèches inversées)
  - 10, 9, 8, 3, 2, 11, 7, 5
  - ...



DAG

image miroir



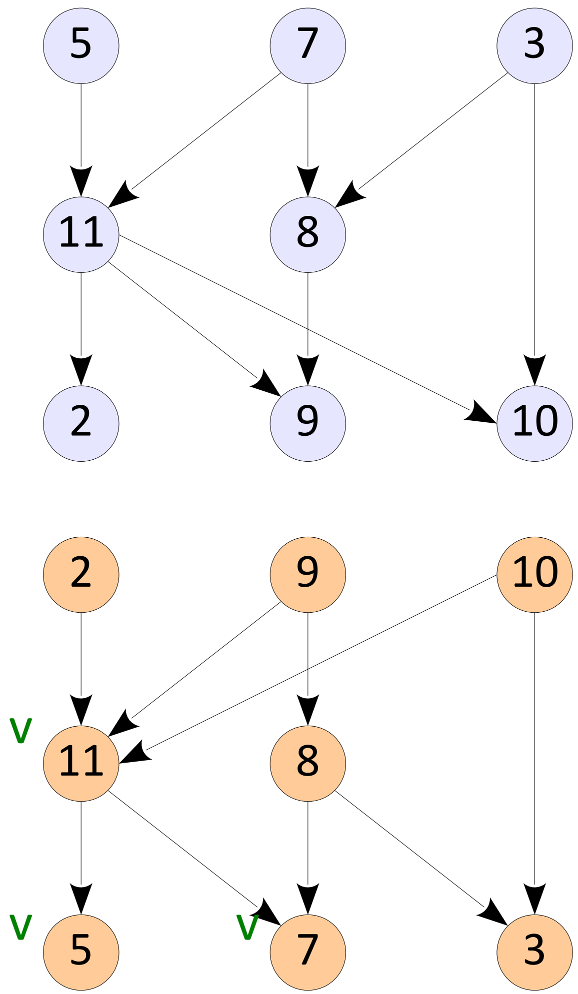
DAG de la relation inverse

Pourquoi est-ce aussi un DAG ?

Preuve par l'absurde

Tri topologique (3)

- Tri topologique ≈ forme de parcours des nœuds du DAG
- Un algorithme :
  - considérer la relation inverse
  - parcours en profondeur d'abord des nœuds non encore visités
  - positionnement d'un drapeau (flag) « déjà visité » après visite
  - liste des nœuds « en remontant » (quand tous les fils sont explorés)
- Ex. 5, 7, 11, 2, 3, 8, 9, 10
- O(*nb nœuds + nb branches*)



DAG de la relation inverse

Compression de données

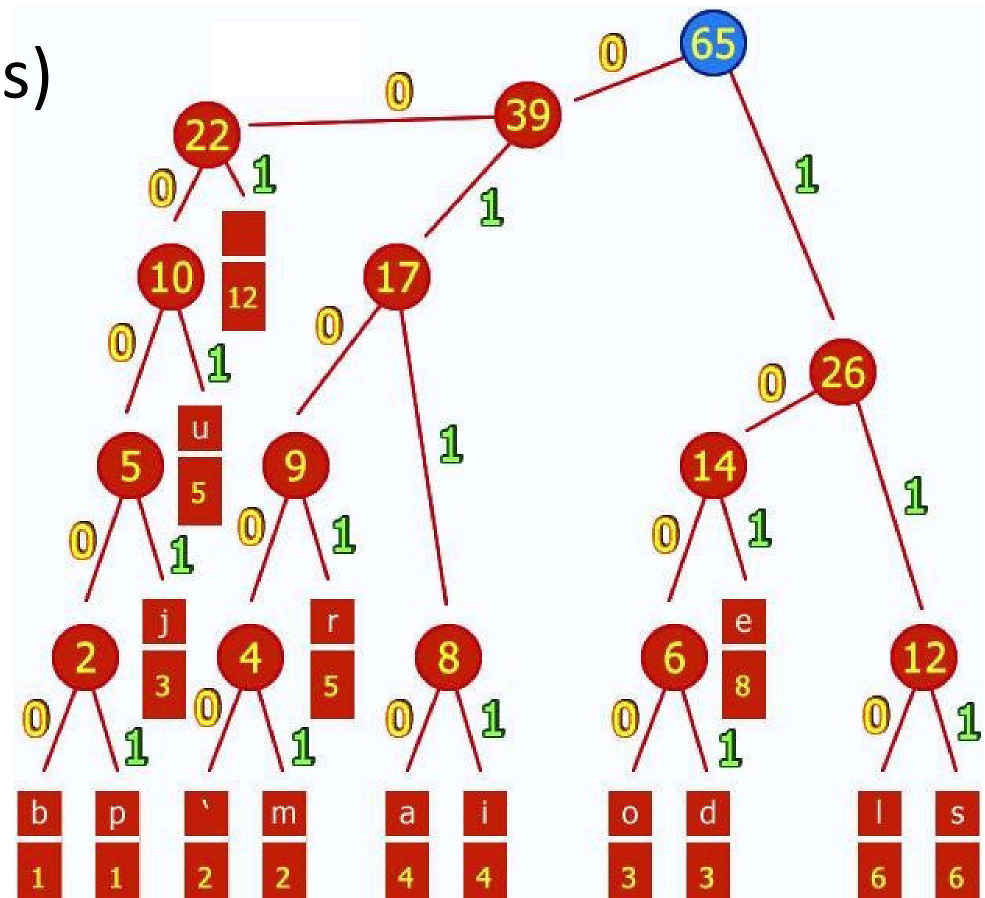
- Donnée :
  - un texte (une suite de caractères)
- Problème :
  - trouver un codage compact du texte
- Une solution :
  - associer un code court aux lettres fréquentes
  - associer un code long aux lettres rares
- Moyen
  - arbre binaire (de Huffman)

Arbre de Huffman (1)

- Arbre binaire
- Info sur feuille :
  - lettre
  - fréquence (≈ nb occurrences)
- Info sur nœud :
  - ensemble de lettres (⇒ alternatives)
  - fréq. ≈ ∑ nb occurrences
- Info sur branche :
  - code 0/1 ≈ choix d'un sous-ensemble de lettres

j'aime aller sur le bord de l'eau
les jeudis ou les jours impairs

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12



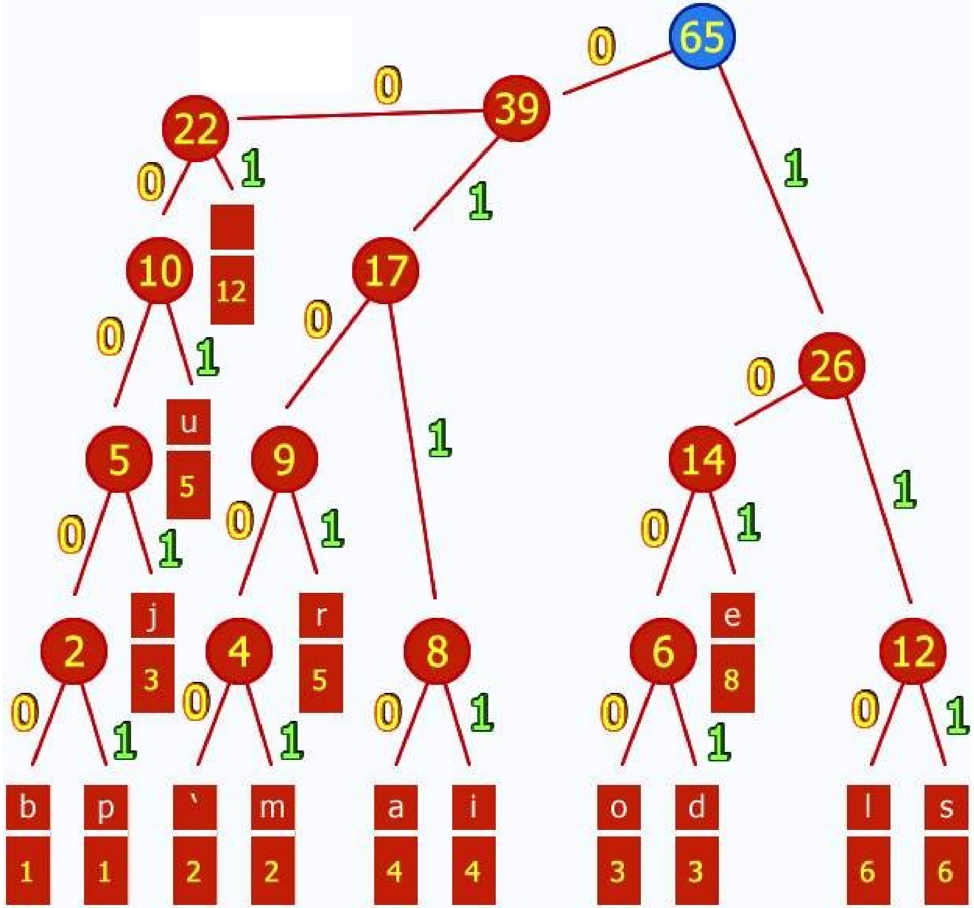
© Philippe Proulx

Arbre de Huffman (2)

- Codage du téléphone
  - 15, 17, 18, 1013, 1023
  - 0123456789, 0631415927
  - 00 1 212 229 2560,
    00 39 010 869 8721
- Encodage avec arbre
  - e : 101
  - i : 0111
  - d : 1001
  - p : 000001
  - pied : 000001011111011001
- Idem coder et décoder
  (non ambigu)

j'aime aller sur le bord de l'eau
les jeudis ou les jours impairs

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12



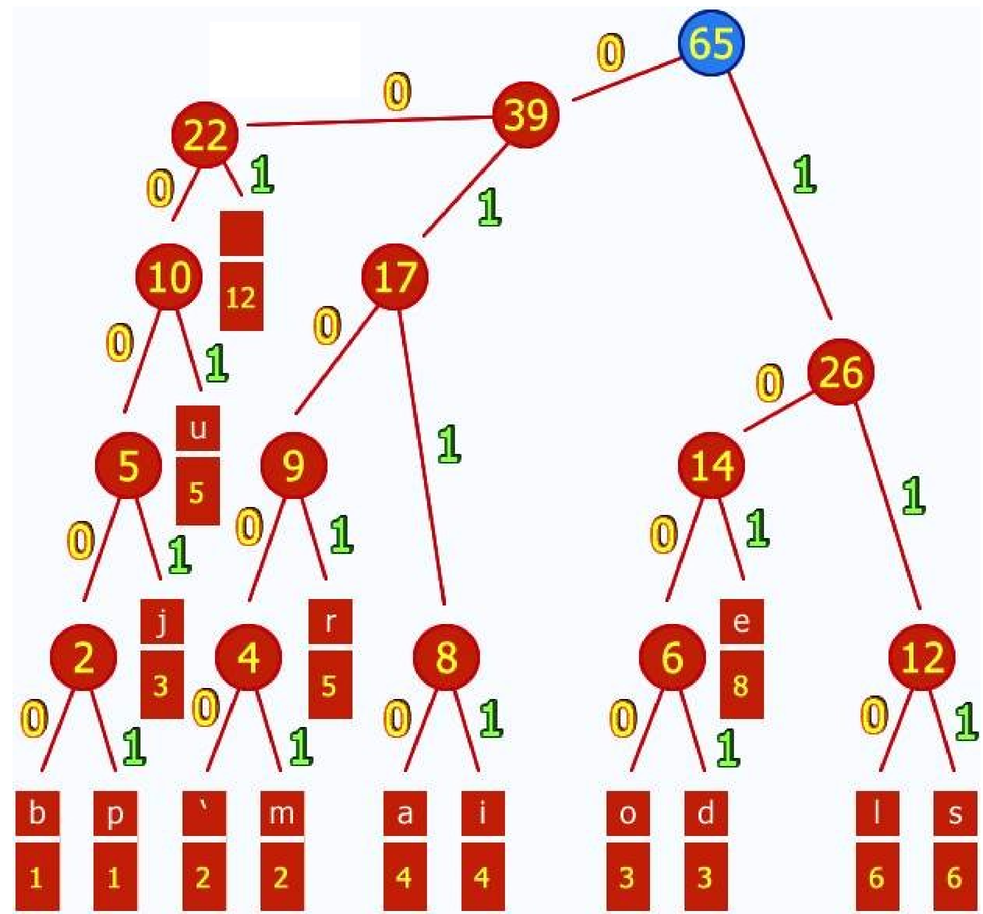
© Philippe Proulx

Arbre de Huffman (3)

- Construction bottom-up (de bas en haut) itérative
  - on prend les 2 lettres ou groupes de lettres les moins fréquents
  - on les associe : on crée un nœud père qui les rassemble (ce nouveau groupe est plus fréquent au prix d'un code plus long)
  - on recommence jusqu'à ce qu'il ne reste qu'un groupe

j'aime aller sur le bord de l'eau les jeudis ou les jours impairs

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12



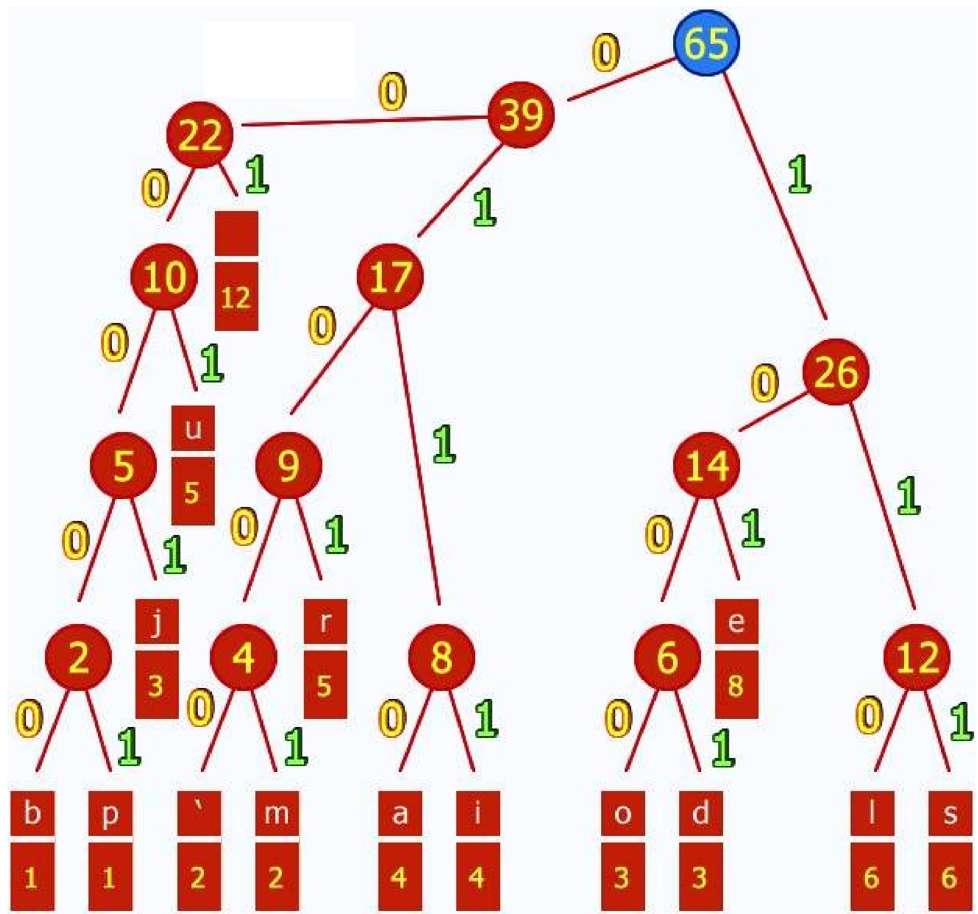
© Philippe Proulx

Arbre de Huffman (4)

- Arbre non unique
- Arbre spécifique aux fréquences dans le texte
  - code = arbre + codage
  - ou bien arbre fixe, choisi par convention (ex. issu d'un grand texte dans une langue donnée)
- Taux de compression en français ≈ 50%

j'aime aller sur le bord de l'eau
les jeudis ou les jours impairs

b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12



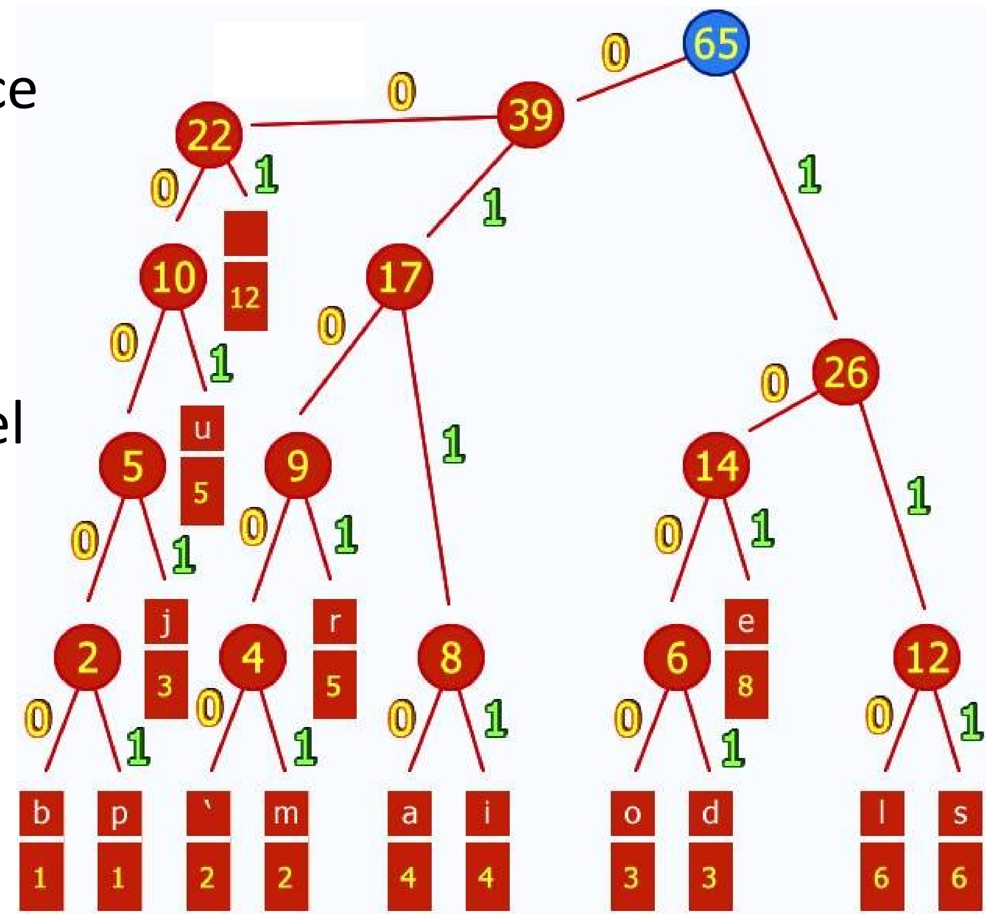
© Philippe Proulx

Arbre de Huffman (5)

- Optimal au sens du codage par symbole : lgr ≈ entropie du texte
- Multiples variantes :
  - ex. prédiction par reconnaissance partielle (PPM) : compression probabiliste en fonction du contexte, OK flux de caractères
  - ex. codage adaptatif (OK flux de caractères) : arbre conventionnel mis à jour dynamiquement
- Meilleurs algorithmes :
  - encodage des sous-chaînes

j'aime aller sur le bord de l'eau
les jeudis ou les jours impairs

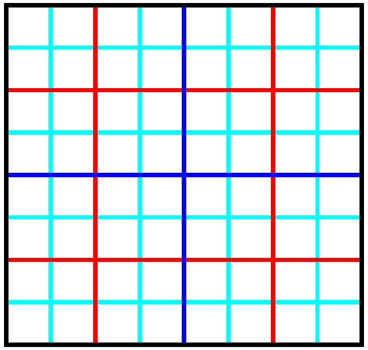
b	p	`	m	j	o	d	a	i	r	u	l	s	e	
1	1	2	2	3	3	3	4	4	5	5	6	6	8	12

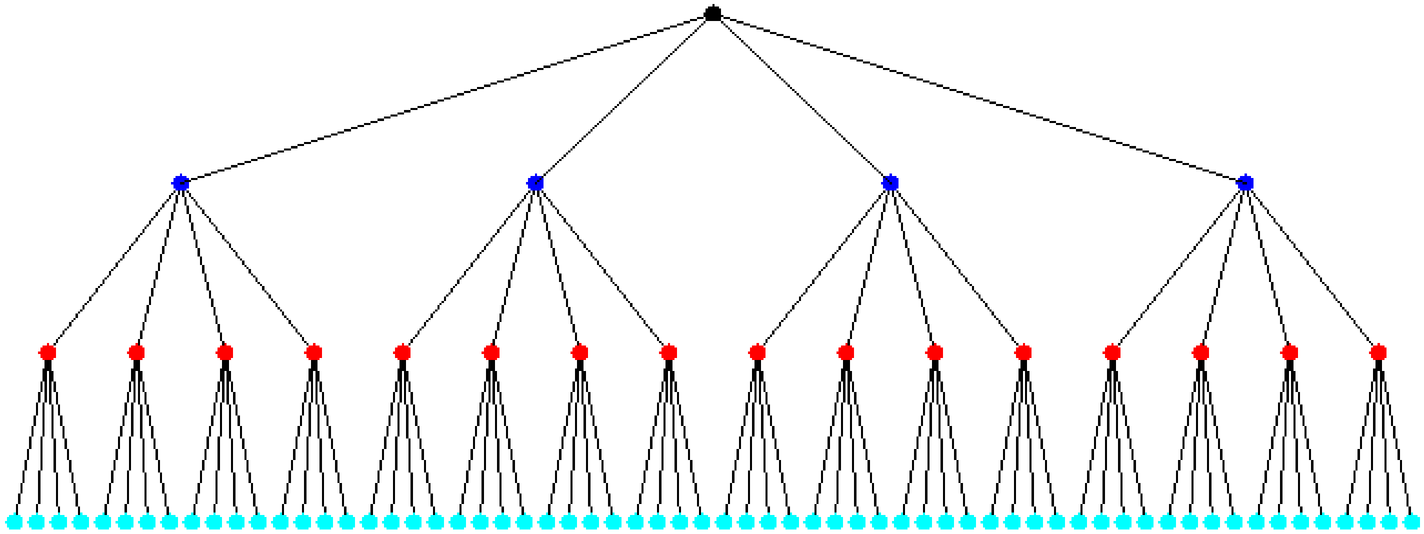


© Philippe Proulx

Partitionnement récursif multidimensionnel (systématique)

- Ex. dimension 2

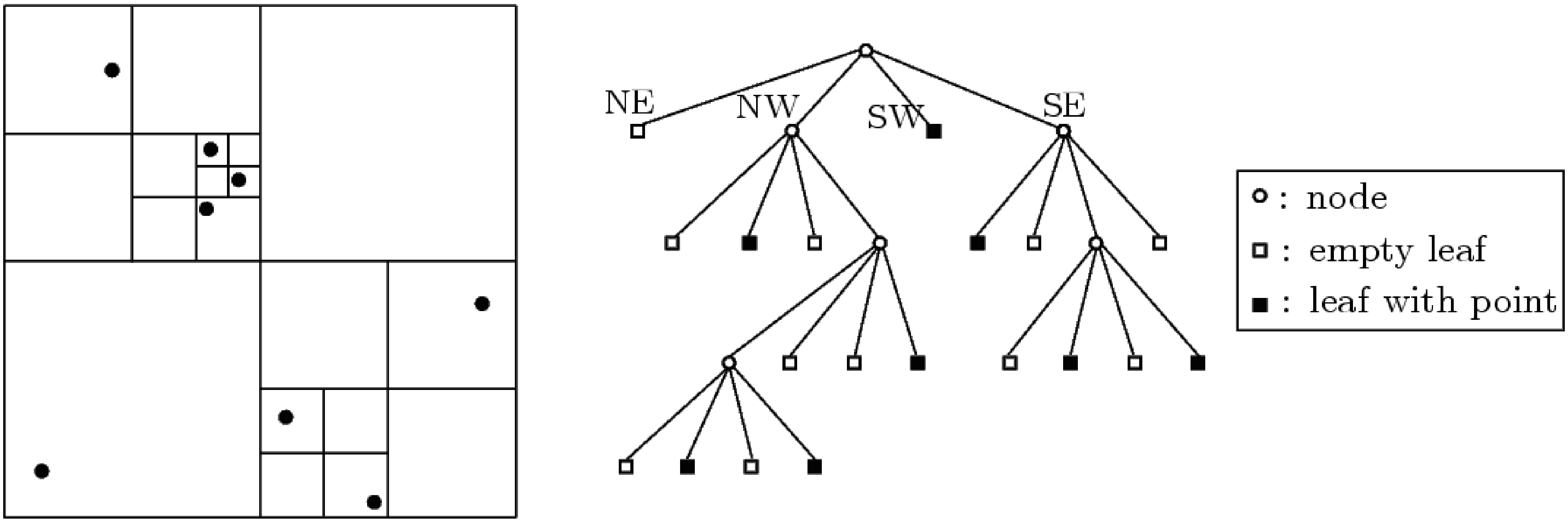




© James Demmel

Partitionnement récursif multidimensionnel (optimisé)

- Au plus 1 point par région
- Au moins 1 point dans une sous-région



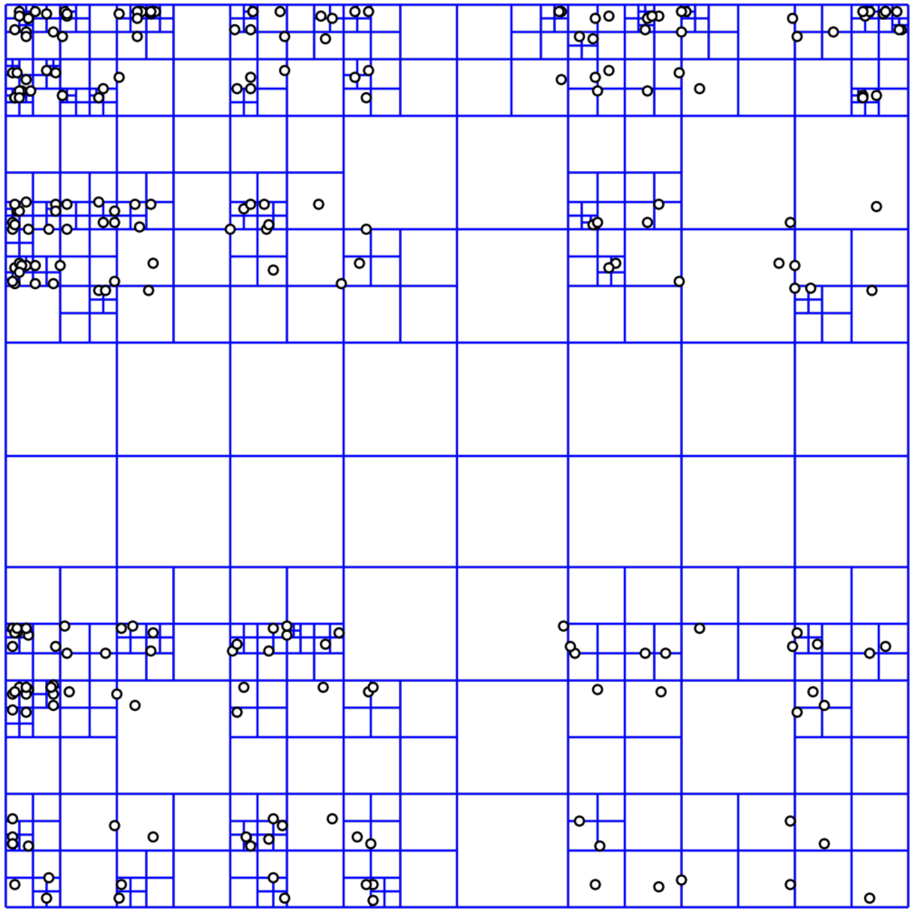
© Hugo Ledoux

Quadtree (ou Q-tree)
[arbre quaternaire]

- Nœud : exactement 4 fils
- Feuille : 0 ou 1 point
- Obtenu par partition récursive d'un espace à 2 dimensions en 4 quadrants

  [= généralisation en 2D d'un arbre binaire en 1D]

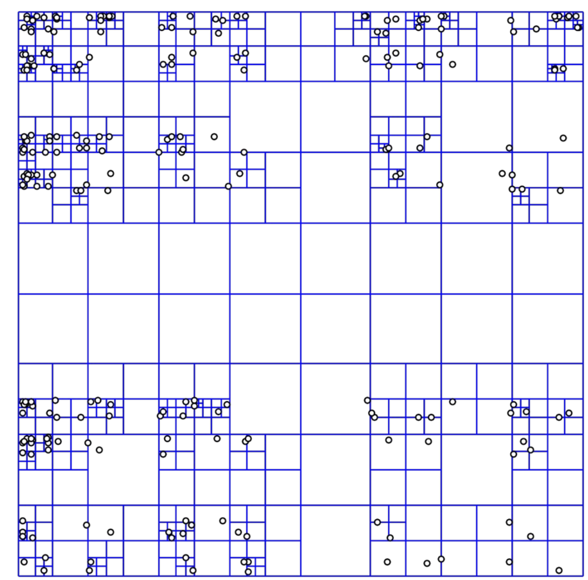
- Terminologies :
  - nord-ouest, sud-est...
  - ou haut-gauche, bas-droite...



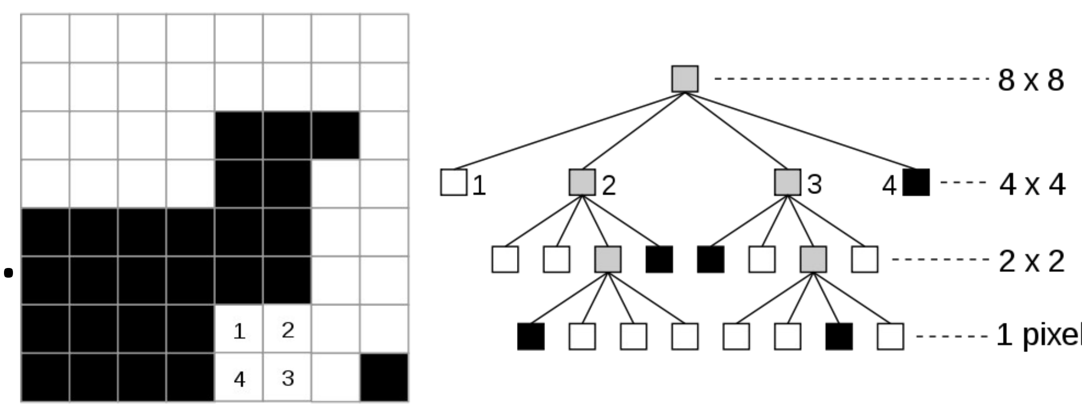
© David Eppstein

Quadtree

- Applications
  - stockage matrice creuse, feuille Excel
    - 0 ou vide presque partout
  - calcul d'un maximum de couverture
    - plus grand ensemble de formes géométriques sans recouvrement
    - ex. design de circuits intégrés, de cartes
  - indexation spatiale
  - détection de collision
  - compression d'images
  - ...
- Nombreuses variantes...



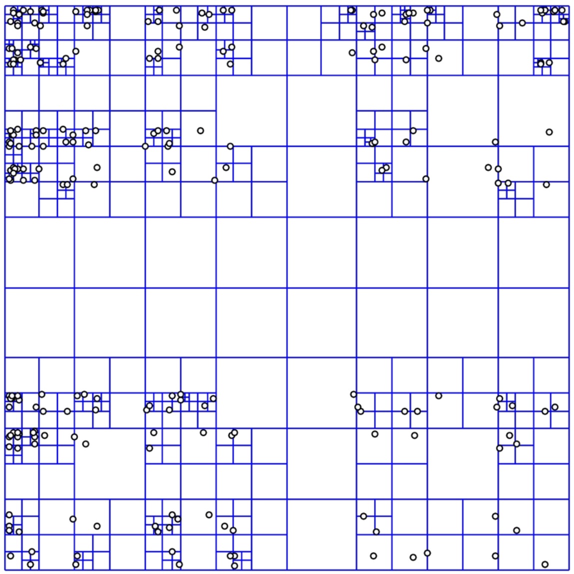
© David Epstein



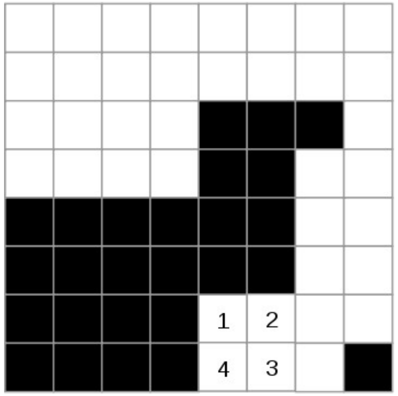
© Wojciech Muła

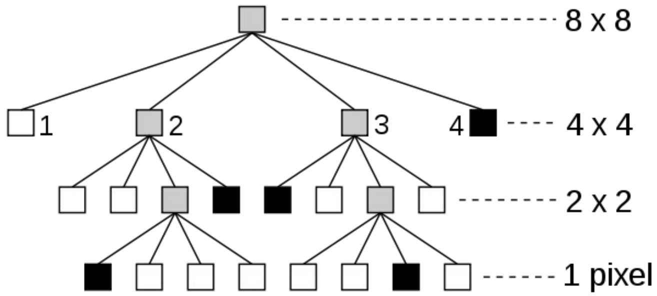
Quadtree

- Applications
  - stockage matrice creuse, feuille Excel
    - 0 ou vide presque partout
  - calcul d'un maximum de couverture
    - plus grand ensemble de formes géométriques sans recouvrement
    - ex. design de circuits intégrés, de cartes
  - indexation spatiale
  - détection de collision
  - compression d'images
  - ...
- Nombreuses variantes...



© David Eppstein

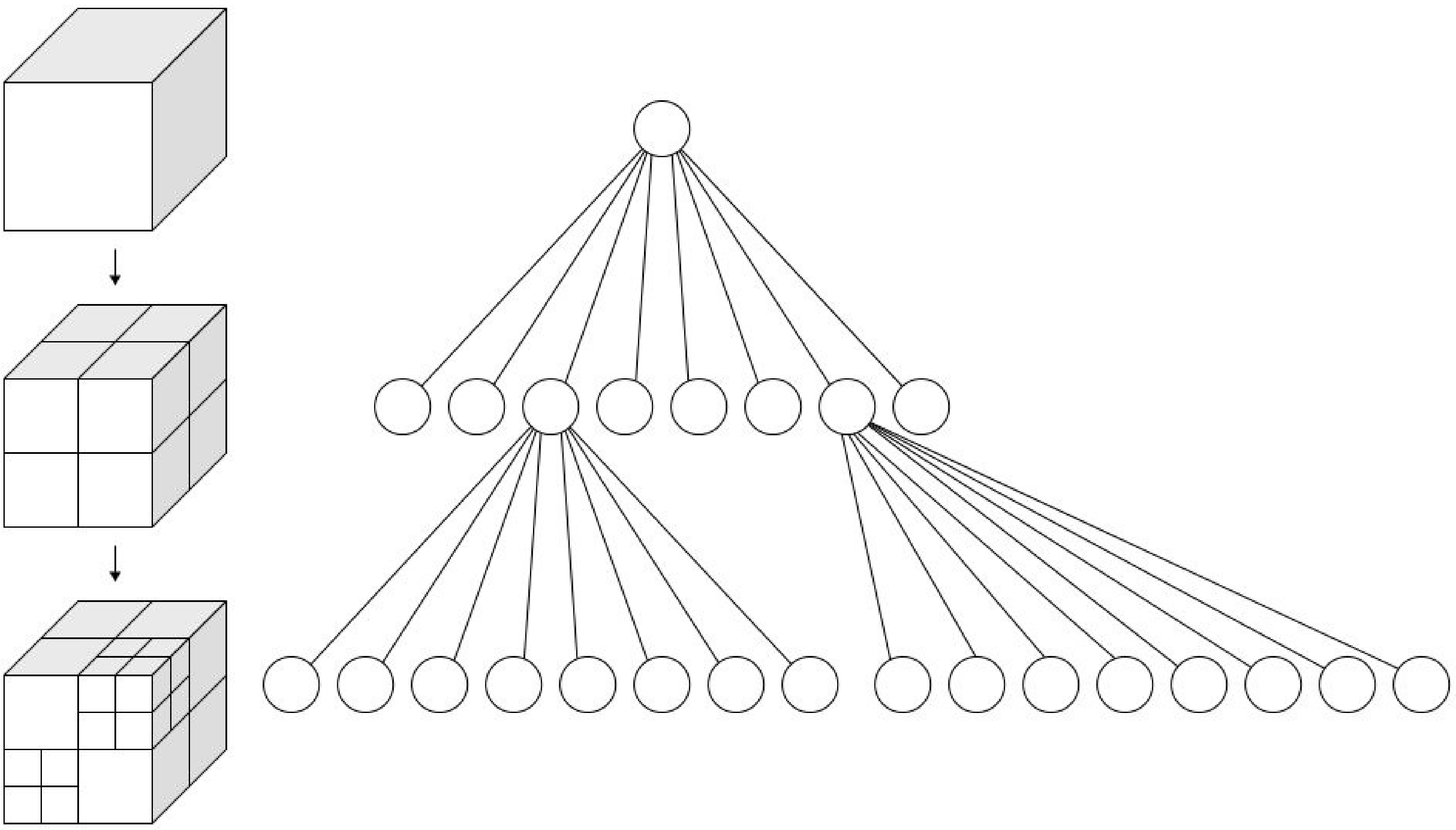




© Wojciech Muła

Octtree

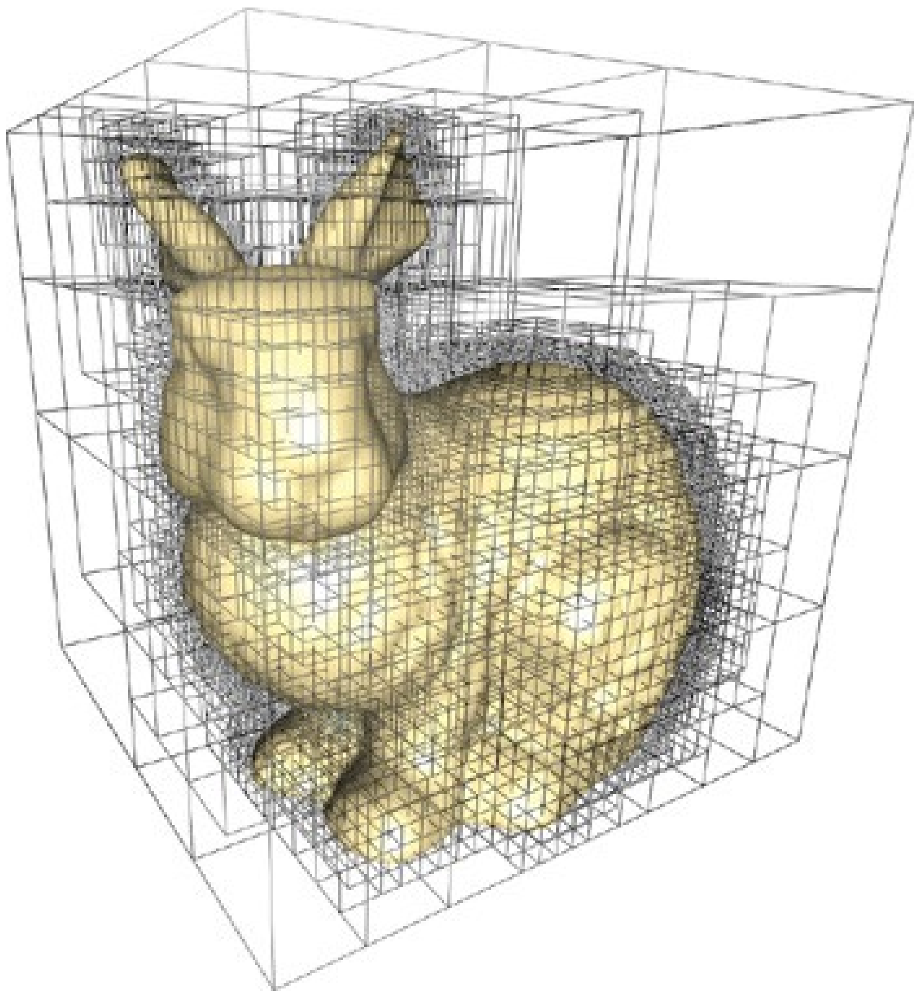
- Idem en dimension 3



© White Timberwolf

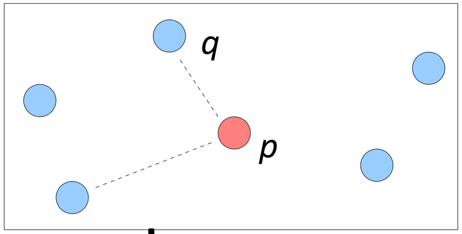
Octtree

- Idem en dimension 3 : ex. voxels



© Lefebvre et al.

Recherche de plus proche(s) voisin(s) [nearest neighbor, NN] et variantes



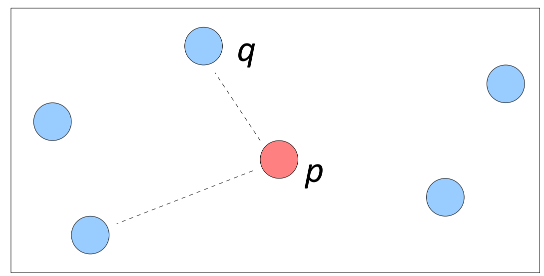
- **Plus proche voisin :**
  - ensemble de n points donnés dans $\mathbb{R}^k$
  - pb : soit un point p, trouver le point q le plus proche (ou les m plus proches)

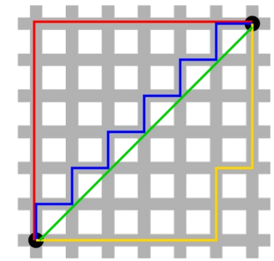
N.B. pas nécessairement unique(s)

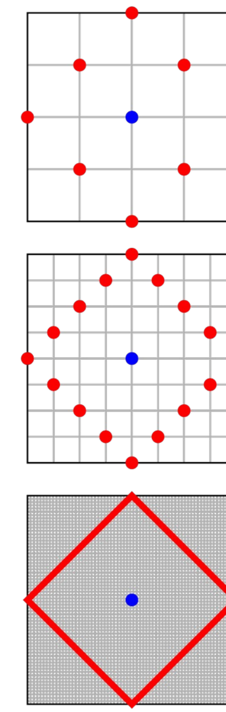
- **Proches voisins :**
  - **exacts :** trouver un/tous les points q tq $d(p,q) \leq r$ donné
  - **approchés :** trouver un/tous les points q tq $d(p,q) \leq (1+\varepsilon)\ d(p,p')$ où p' est le plus proche voisin de p
- **Jointure spatiale :**
  - pb : étant donnés deux sous-ensembles P et Q des n points, trouver les points $(p,q) \in P \times Q$ tq $d(p,q) \leq r$ donné

Recherche de plus proche(s) voisin(s) [nearest neighbor, NN] et variantes

- Variantes de distance (espace métrique) :
  - euclidienne (L_2)
  - Manhattan (L_1)
  - L_p





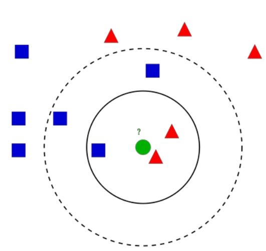


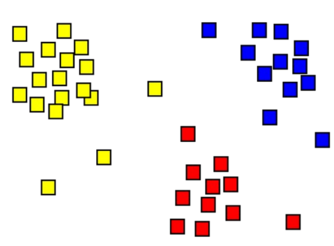
- Variantes d'espace : continu vs discret
- Variantes selon la dimension k
- Variantes d'accès aux données
  - en mémoire vive (RAM) : lecture par octet/entier/flottant
  - stockage de masse (disque dur) : lecture par blocs (R-tree)
    - base de données, systèmes de fichiers
- ...

© Psychonaut, Qef

Quelques applications de la recherche de plus proche(s) voisin(s)

- Simul. physique : interaction jusqu'à distance d bornée
- Jeux vidéo : collision entre objets
- Système d'info. géographique (SIG) : ex. ville la + proche
- Classification : classe la + proche d'une instance donnée
- Reconnaissance de motifs (patterns) : pattern le + proche
- Recherche d'images par le contenu : les + ressemblantes
- Détection de plagiat : documents similaires
- Système de recommandation : qui a des goûts voisins ?
- Partitionnement de données (clustering)
- Séquençage d'ADN
- …



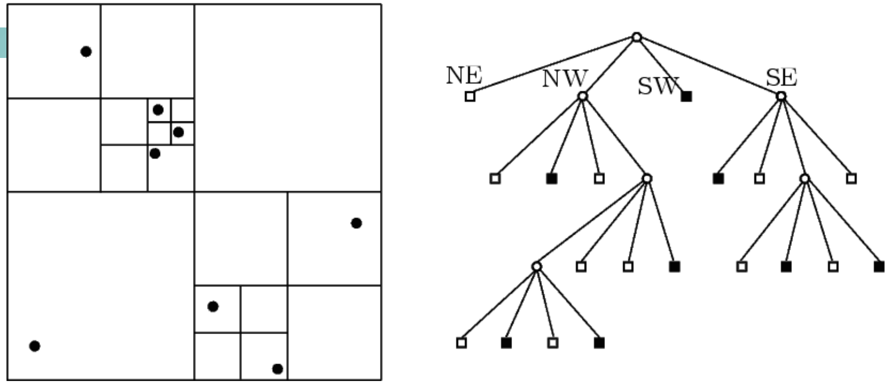


© Wgabrie, CWenger

Propriétés de la recherche de plus proche voisin

- Recherche séquentielle (test pour chaque point)
  - temps : $O(n)$
- Partitionnement récursif de l'espace (→ divers arbres)
  - temps : $O(\log n)$ en moyenne, parfois $O(n)$ dans le pire cas
- Variante : ajout/suppression « dynamique » de points
  - coût de réindexation → compromis (en temps et en espace) nb d'accès / nb de modifications
- Variante : recherche approchée (rapide, ratés possibles)
  - réduction de dimension probabiliste
    - ex. locality sensitive hashing (LSH)

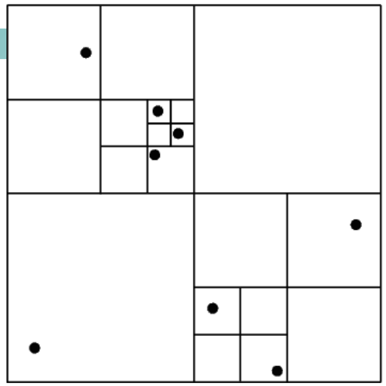
Proches voisins en 2D avec quadtree : Construction

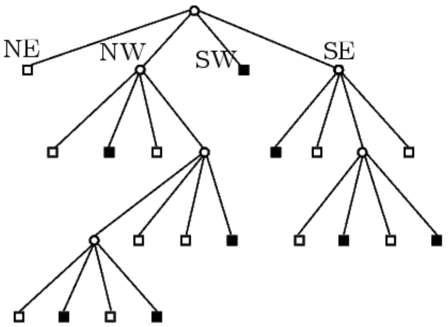


© Hugo Ledoux

- Hypothèse
  - bornes dans l'espace connues (min-max des coordonnées)
- Construction
  - créer une feuille vide (qui représente tout l'espace borné)
  - insérer successivement chaque point
- Insérer un point
  - descendre dans l'arbre en découpant récursivement ses coordonnées
    - jusqu'à atteindre une feuille
  - si la case est vide
    - y mettre le point
  - si la case est occupée par un point
    - la redécouper récursivement (créant des nœuds) jusqu'à ce que
      - les 2 points soient dans 2 cases distinctes

Proches voisins en 2D avec quadtree : Construction



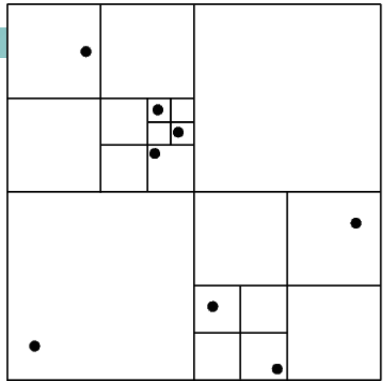


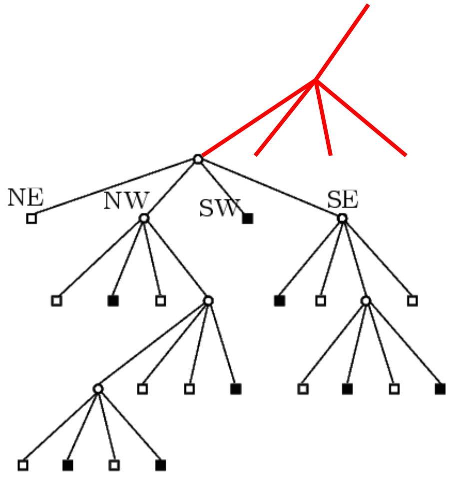
© Hugo Ledoux

- Hypothèse
  - bornes dans l'espace connues (min-max des coordonnées)
- Construction
  - créer une feuille vide (qui représente tout l'espace borné)
  - insérer successivement chaque point
- Insérer un point
  - descendre dans l'arbre en découpant récursivement ses coordonnées
    - jusqu'à atteindre une feuille
  - si la case est vide
    - y mettre le point
  - si la case est occupée par un point
    - la redécouper récursivement (créant des nœuds) jusqu'à ce que
      - les 2 points soient dans 2 cases distinctes

Et si les bornes des coordonnées sont mal/pas connues ?

Proches voisins en 2D avec quadtree : Construction





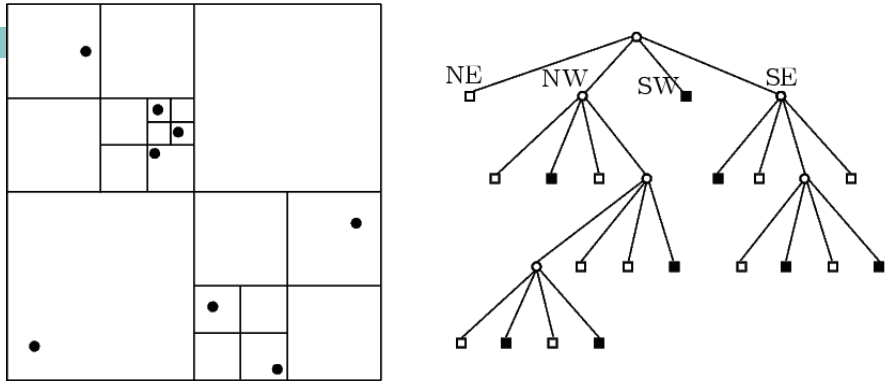
© Hugo Ledoux

- Hypothèse
  - bornes dans l'espace connues (min-max des coordonnées)
- Construction
  - créer une feuille vide (qui représente tout l'espace borné)
  - insérer successivement chaque point
- Insérer un point
  - descendre dans l'arbre en découpant récursivement ses coordonnées jusqu'à atteindre une feuille
  - si la case est vide
    - y mettre le point
  - si la case est occupée par un point
    - la redécouper récursivement (créant des nœuds) jusqu'à ce que les 2 points soient dans 2 cases distinctes

Et si les bornes des coordonnées sont mal/pas connues ?

☛ l'ancienne racine devient le nœud d'une nouvelle racine

Proches voisins en 2D avec quadtree : Construction



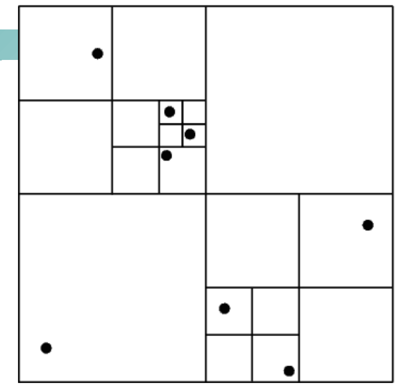
© Hugo Ledoux

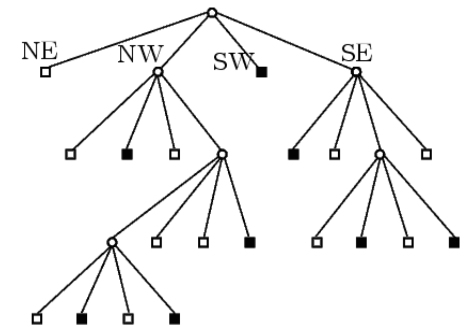
- Hypothèse
  - bornes dans l'espace connues (min-max des coordonnées)
- Construction

  créer une feuille vide (qui représente tout l'espace borné)
  insérer successivement chaque point
- Insérer un point

  descendre dans l'arbre en découpant récursivement ses coordonnées
  jusqu'à atteindre une feuille
  si la case est vide
  y mettre le point
  si la case est occupée par un point différent sinon ?
  la redécouper récursivement (créant des nœuds) jusqu'à ce que
  les 2 points soient dans 2 cases distinctes

Proches voisins en 2D avec quadtree : Construction





© Hugo Ledoux

- Hypothèse
  - bornes dans l'espace connues (min-max des coordonnées)
- Construction

  créer une feuille vide (qui représente tout l'espace borné)
  insérer successivement chaque point
- Insérer un point

  descendre dans l'arbre en découpant récursivement ses coordonnées
      jusqu'à atteindre une feuille
  si la case est vide
      y mettre le point
  si la case est occupée par un point différent
      la redécouper récursivement (créant des nœuds) jusqu'à ce que
          les 2 points soient dans 2 cases distinctes

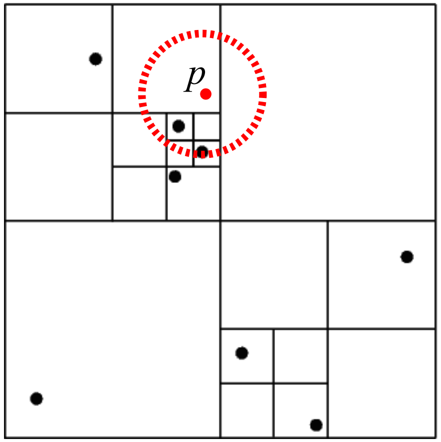
$\boxed{\text{sinon : boucle } \infty}$

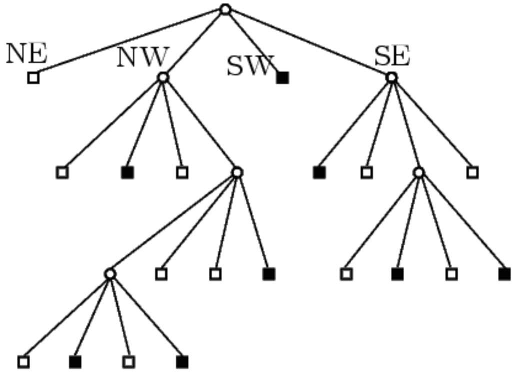
Proches voisins en 2D avec quadtree : Recherche à distance donnée

- Rechercher tous les voisins à distance r d'un point p

partir du noeud t racine de l'arbre
pour chaque fils f de t
    si f est une feuille, examiner/traiter l'éventuel point q de f :
        si $d(p,q) \leq r$ alors mémoriser q
    sinon, si la case f intersecte le disque de rayon r centré en p :
        rechercher récursivement les voisins dans f

Gain de temps avec la distance L_2 :
- ne pas calculer la racine carré $((x_p - x_q)^2 + (y_p - y_q)^2)^{1/2}$
- opérer sur la distance au carré $(x_p - x_q)^2 + (y_p - y_q)^2$ pour toutes les **comparaisons**





© Hugo Ledoux

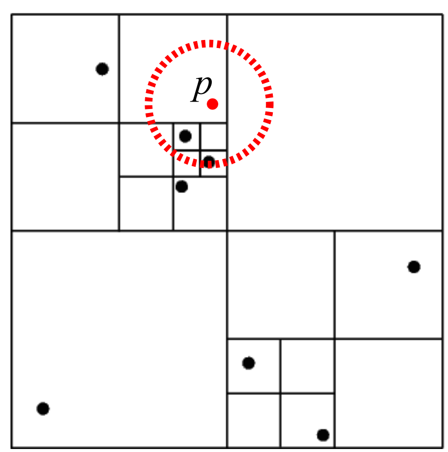
Proches voisins en 2D avec quadtree : Recherche à distance donnée

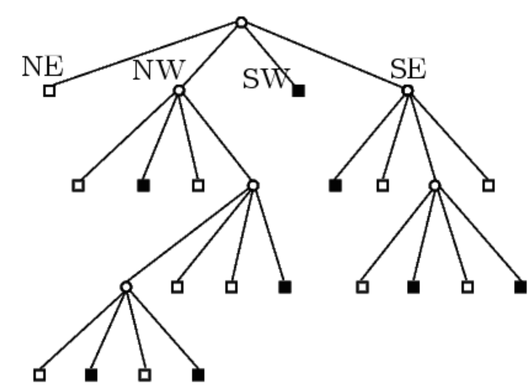
- Rechercher tous les voisins à distance r d'un point p

Variante avec pile explicite (au lieu de la pile d'exécution des appels récursifs)

créer une **pile de nœuds**, initialement vide
mettre la racine sur la pile
répéter tant que la pile n'est pas vide
    extraire un nœud t de la pile
    pour chaque fils f de t
        si f est une feuille, examiner l'éventuel point q de f : $d(p,q) \leq r$?
        sinon, si la case f intersecte le disque de rayon r centré en p
            ajouter f dans la pile

Gain de temps avec la distance L_2 :
- ne pas calculer la racine carré
  $((x_p - x_q)^2 + (y_p - y_q)^2)^{1/2}$
- opérer sur la distance au carré
  $(x_p - x_q)^2 + (y_p - y_q)^2$
  pour toutes les **comparaisons**





© Hugo Ledoux

Proches voisins en 2D avec quadtree : Point le plus proche

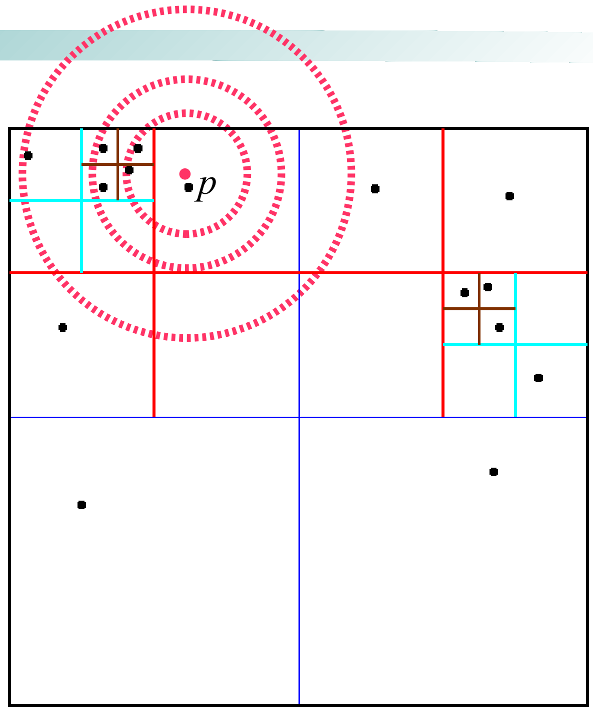
- Trouver le plus proche voisin de p

$r \leftarrow \infty$

rechercher les voisins à distance < r de p

qd un voisin q est trouvé à $d(p,q) < r$

$r \leftarrow d(p,q)$



© James Demmel

Autrement dit

partir du noeud t racine de l'arbre

$r \leftarrow \infty$

pour chaque fils f de t

si f est une feuille, examiner/traiter l'éventuel point q de f :

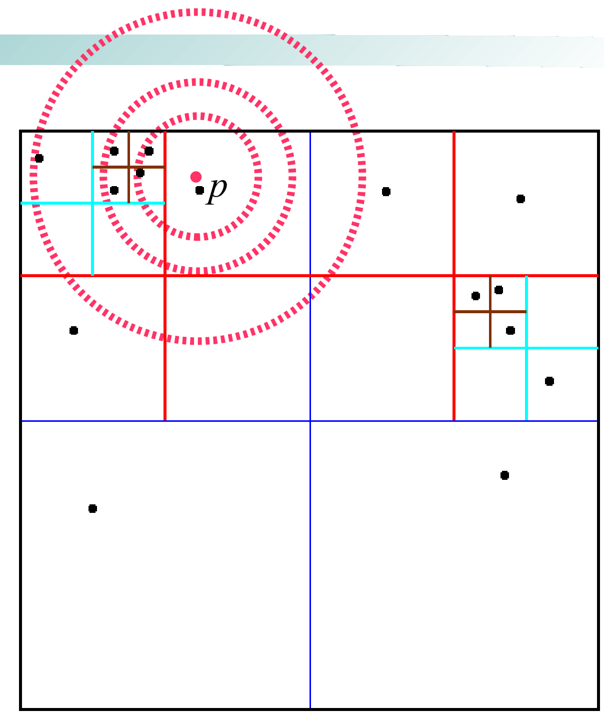
si $d(p,q) < r$ alors mémoriser q et $r \leftarrow d(p,q)$

sinon, si la case f intersecte le disque de rayon r centrée en p :

rechercher récursivement les plus proches voisins dans f

Proches voisins en 2D avec quadtree : Point le plus proche

- Trouver le plus proche voisin de p



$r \leftarrow \infty$
rechercher les voisins à distance $< r$ de p
  qd un voisin q est trouvé à $d(p,q) < r$
    mettre à jour $r \leftarrow d(p,q)$

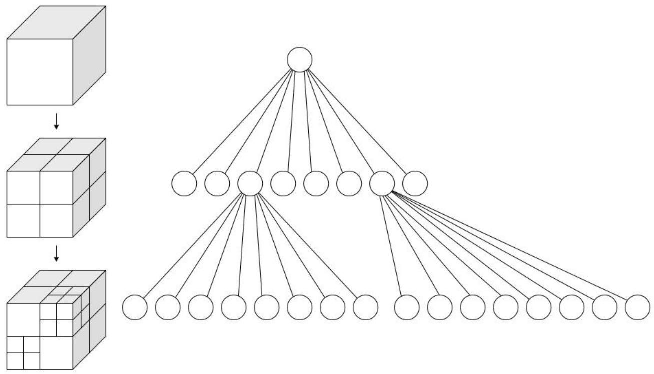
créer une pile de nœuds, initialement vide
mettre la racine sur la pile
$r \leftarrow \infty$
répéter tant que la pile n'est pas vide
  extraire un nœud t de la pile
  pour chaque fils f de t
    si f est une feuille : si f contient q tq $d(p,q) < r$, alors $r \leftarrow d(p,q)$
    sinon, si f intersecte le disque de rayon r centré en p
      ajouter f dans la pile

Variante avec pile explicite (au lieu de la pile d'exécution des appels récursifs)

© James Demmel

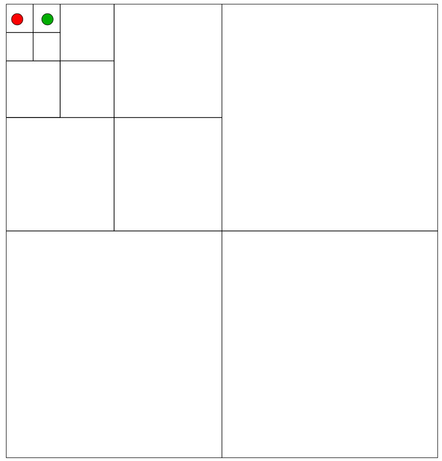
Proches voisins en 2D avec quadtree

- Autres dimensions : idem
  - dimension 3 : octtree
  - dimension k : hypercubes
  - disque de rayon r → boule de rayon r



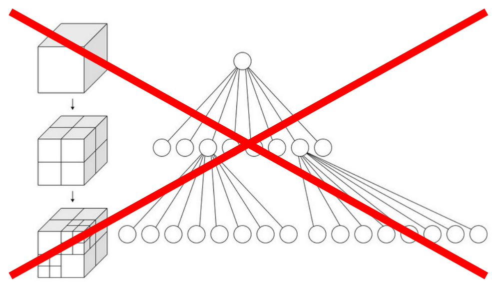
© White Timberwolf

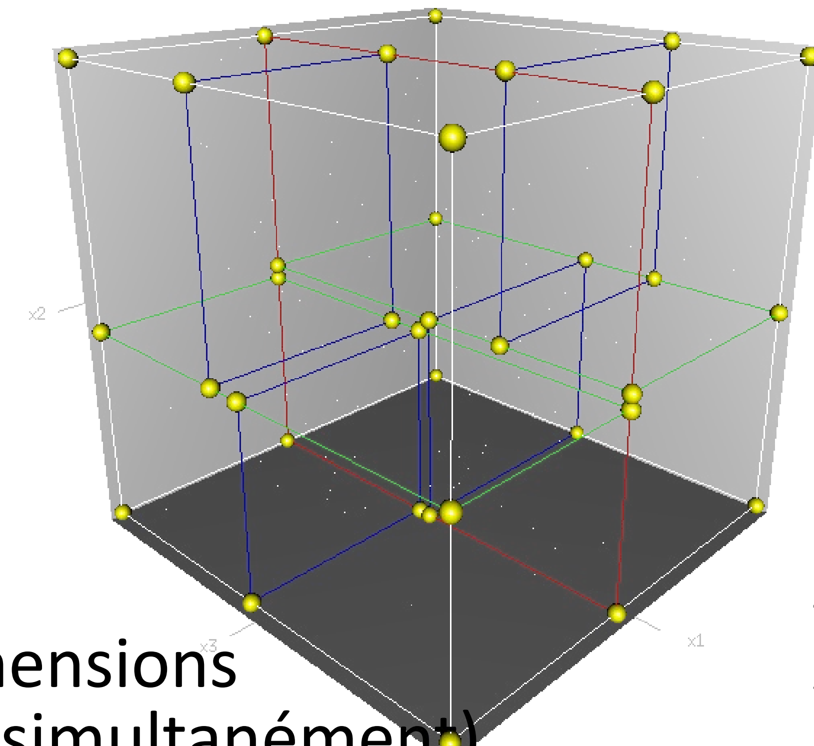
- Quelques faiblesses néanmoins
  - lent si les points sont peu denses (vs un test exhaustif linéaire)
  - profondeur excessive si deux points sont très proches
  - exponentiel en temps et en espace avec la dimension k (facteur 2^k)



*k*d-tree (*k*-dimensional tree)

- Structuration de points en **dimension *k* quelconque**



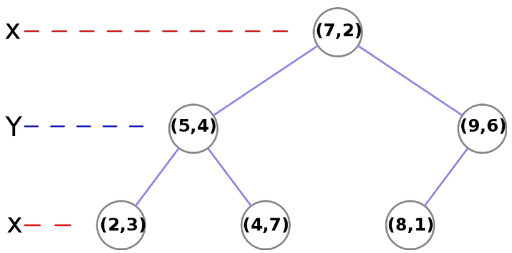


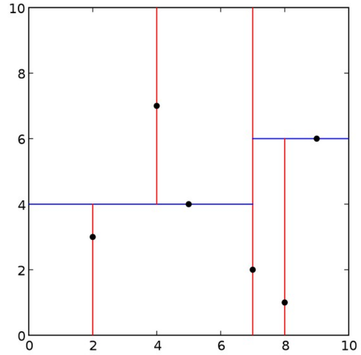
© Benjamin Tyner

- Coupes binaires
  - **adaptatives**
    (pas forcément au milieu)
  - **successives** dans différentes dimensions
    (pas dans toutes les dimensions simultanément)
  - exemple :
    - coupes rouge, vert, bleu
    - cycle sur les dimensions

*k*d-trees équilibrés

© KindDragon33, MYguel

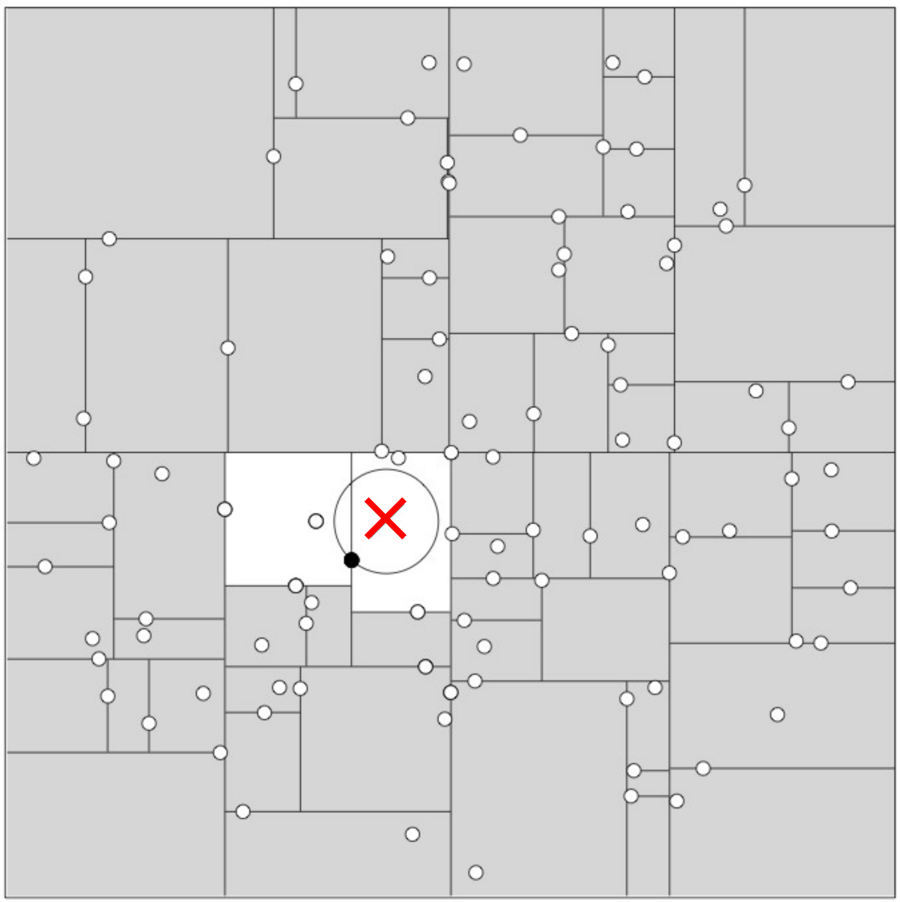


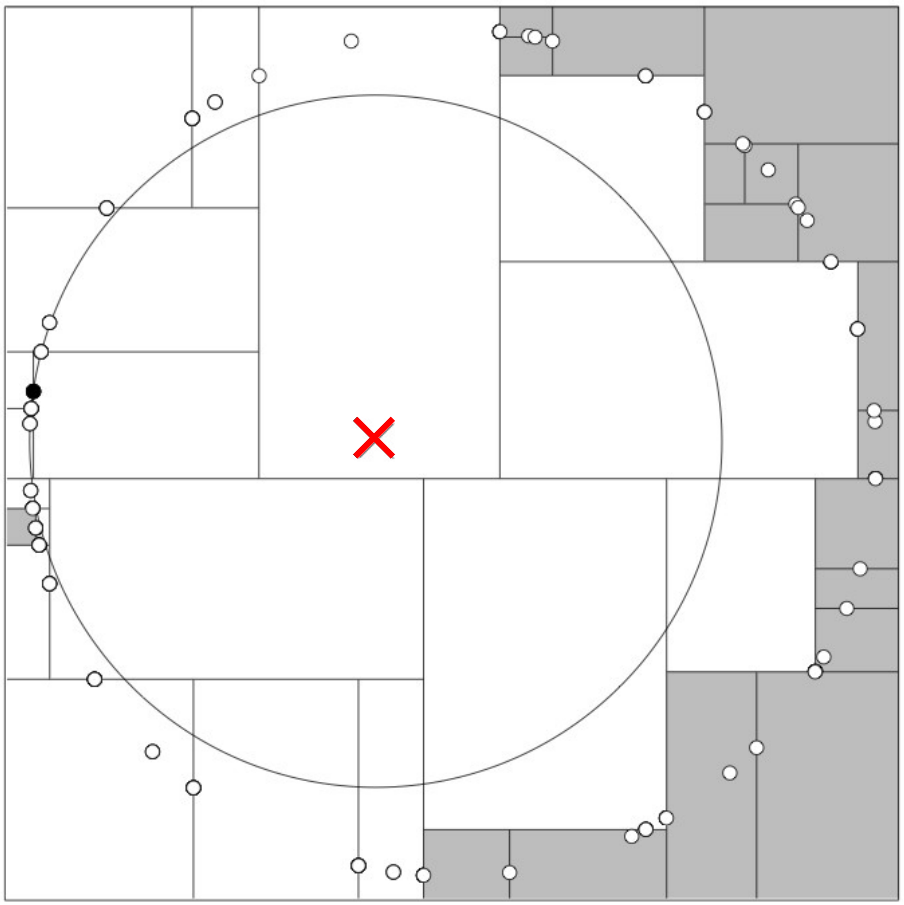


- Coupe (pivot) = pt médian
  [suppose tous les points connus]
- Construction
  - taille : $O(n)$, peu d'espace vide
  - temps : $O(n \log n)$ ou $O(n \log^2 n)$ selon algo. pour médiane
- Temps de recherche du plus proche voisin
  - moyenne : $O(\log n)$ si points distribués uniformément
  - pire cas : $O(k\, n^{1\text{-}1/k})$
- Efficace si $n >> 2^k$, sinon proche du parcours exhaustif
  ☛ profitable en pratique **pour de faibles dimensions**
- Recherche approchée : complexités meilleures

Recherche de proches voisins via *k*d-tree : cas favorable ou défavorable

- Cas favorable = courant, si points aléatoires
- Cas défavorable, si biais sur les données

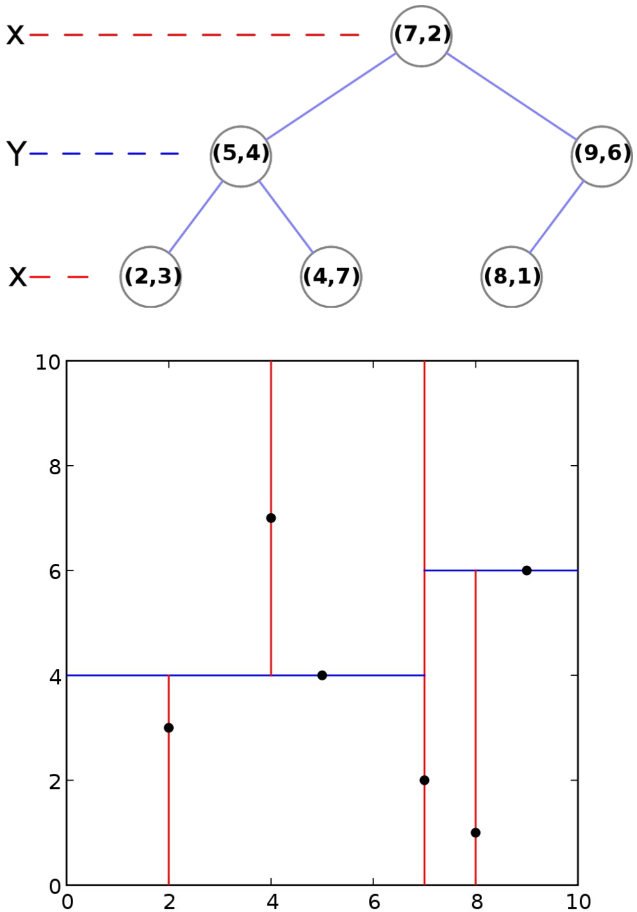




© Andrew Moore

*k*d-tree équilibré (ou presque)

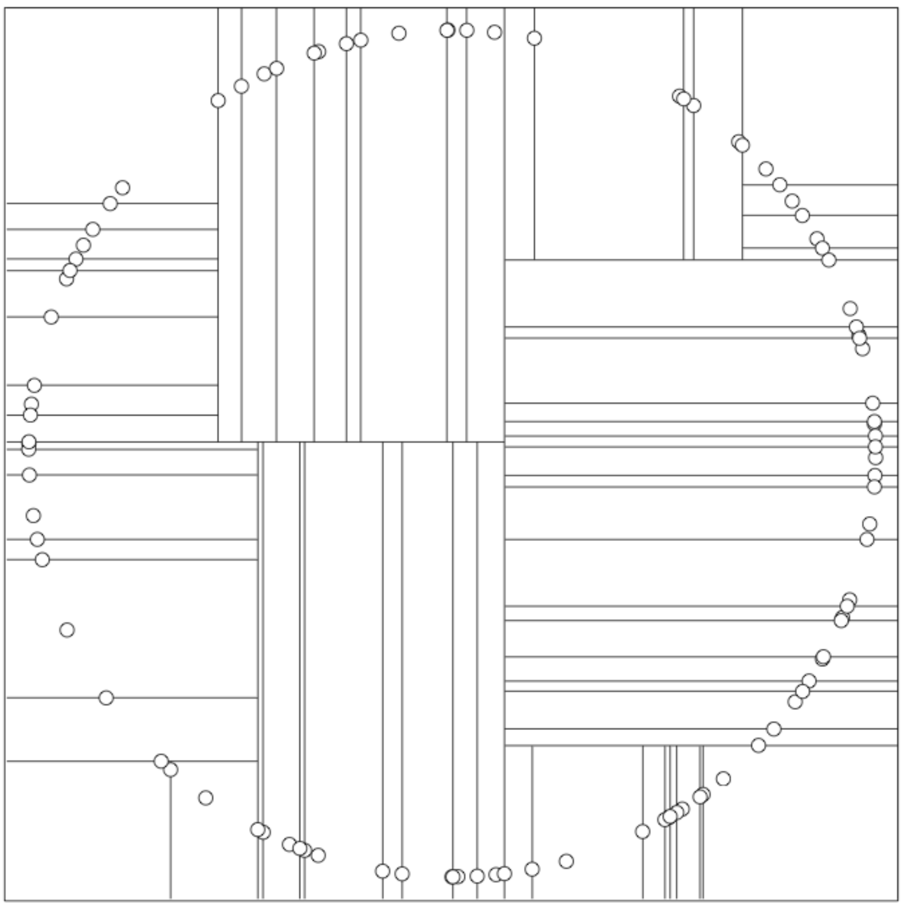
© KindDragon33, MYguel



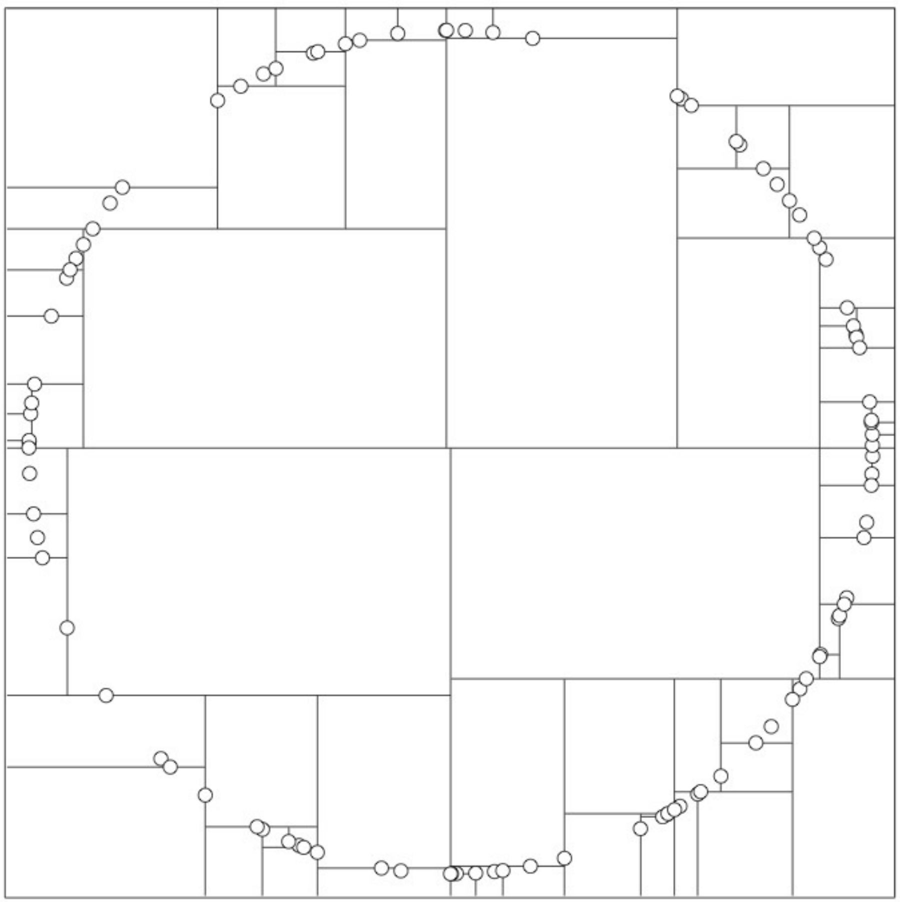
- Médiane exacte
  - via un tri → $O(n \log n)$
  - quickselect → $O(n)$, pire cas $O(n^2)$
- Médiane approchée → $O(n)$ pire cas
  - « médiane des médianes » :
    - diviser en $n/5$ groupes de 5 et prendre la
    - médiane de chaque groupe → $n/5.O(1)$
    - itérer sur les $n/5$ médianes ...→... $O(n)$
      ☛ médiane finale entre 30% et 70%
  - aléatoire :
    - médiane d'un petit nombre m de points tirés au hasard → $O(m)$
    - pivot qcq → $O(1)$, recherche efficace malgré risque de déséquilibre

Influence du choix de la dimension et du pivot selon la distribution des points

ex. **médiane** de la dimension la plus étendue → bon équilibre



ex. **milieu** de la dimension la plus étendue → léger déséquilibre



© Andrew Moore

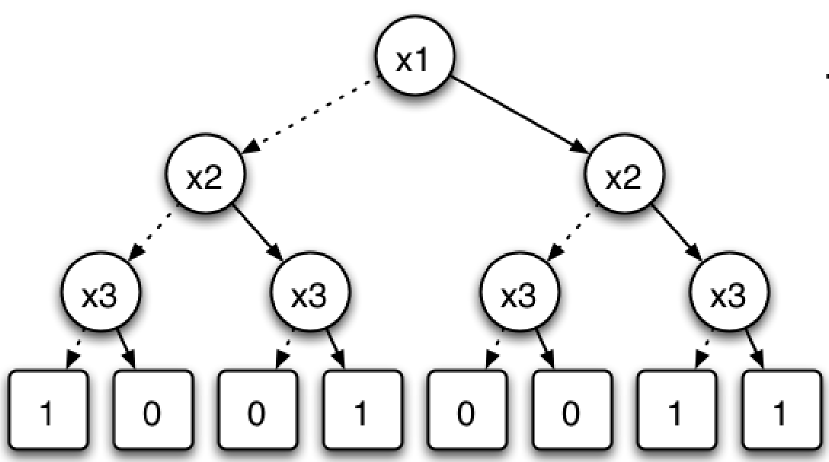
Autour des *k*d-trees

- Nombreuses variantes
  - choix de la dimension à couper
  - choix du point pivot (→ pas systématiquement équilibré)
  - ...
- Ensemble de points statique ou dynamique ?
  - ensemble statique et beaucoup de recherches
    → construire un *k*d-tree optimal : $O(\mathrm{n} \log n)$
  - ensemble dynamique
    - insertion, suppression d'un point en $O(\log n)$
    - préservation de l'équilibre = délicat

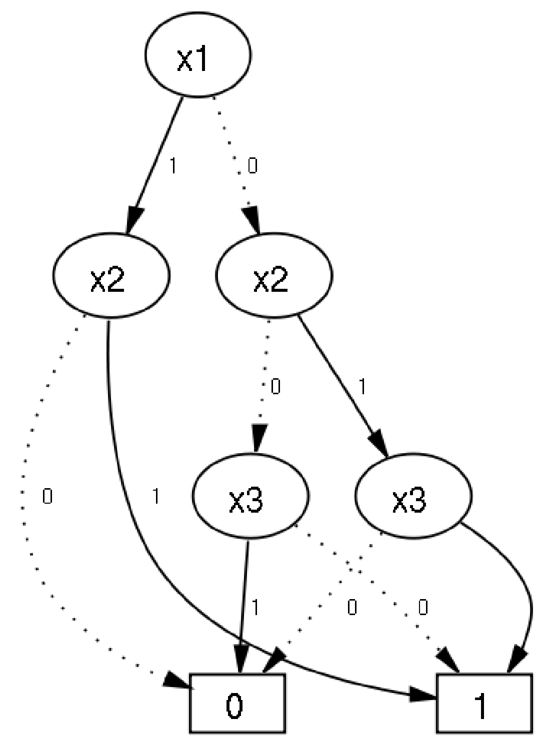
Diagramme de décision binaire (BDD)
[binary decision diagram]

- Fonction logique de *k* variables

  f : {true, false}^{k} → {true, false}
- Représentation avec un DAG
- BDD : très puissant, très efficace



x1	x2	x3	*f*
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

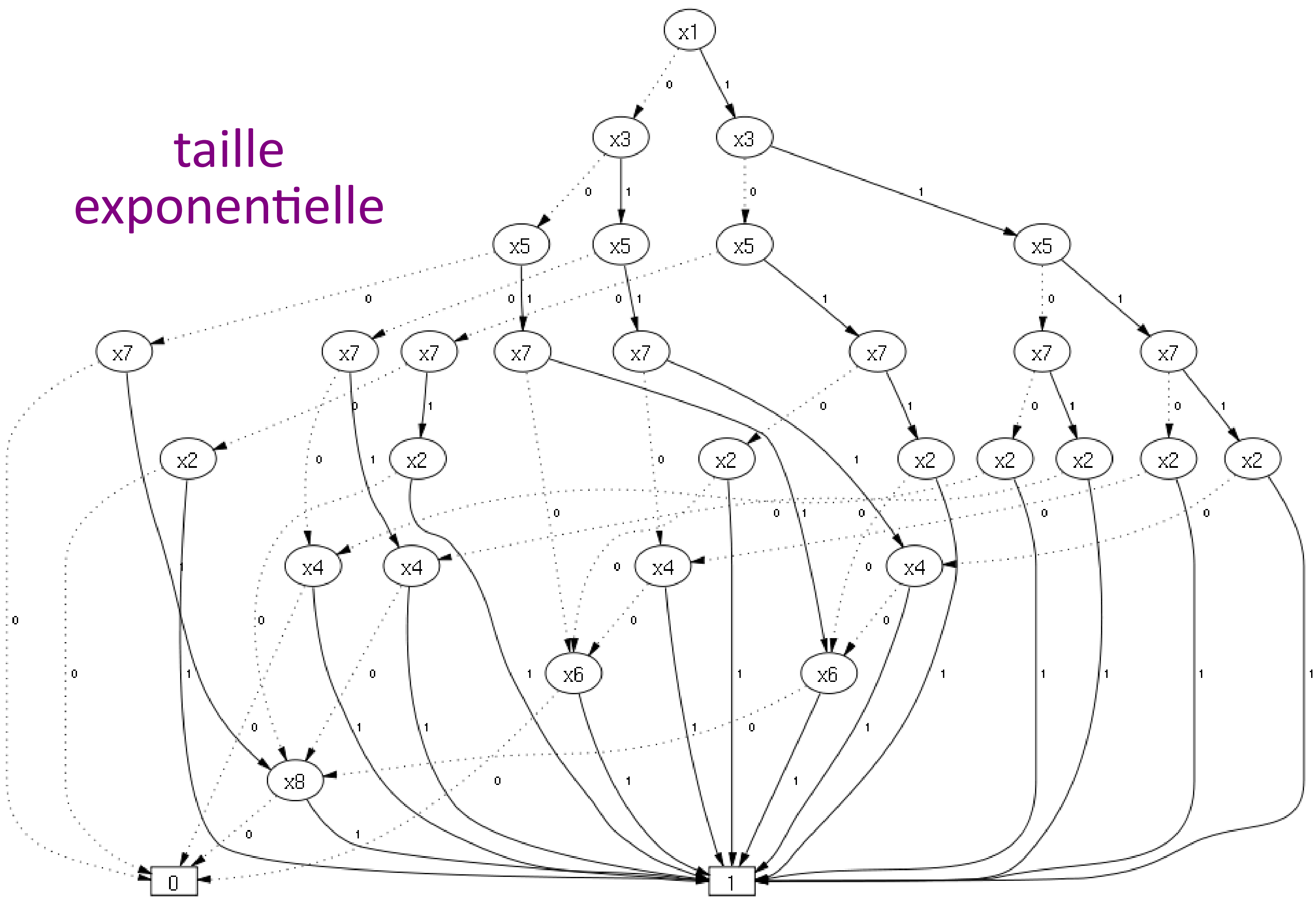


© Dirk Beyer

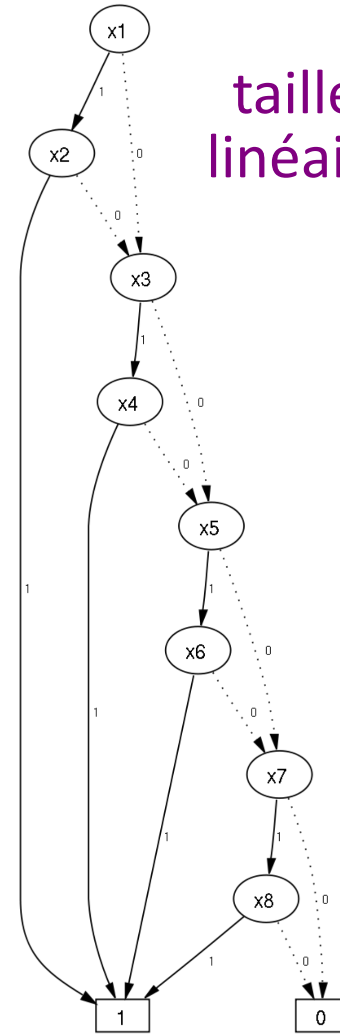
Diagramme de décision binaire (BDD)

- Impact de l'ordre des variables (pour même fonction)

taille exponentielle



taille linéaire



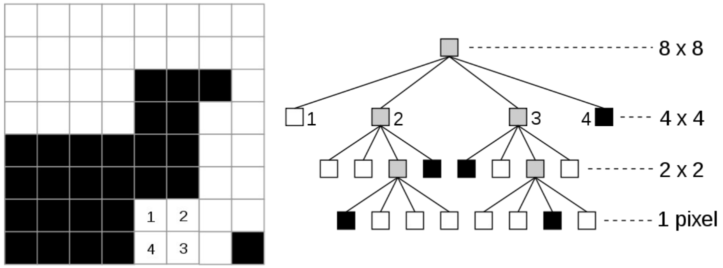
© Dirk Beyer

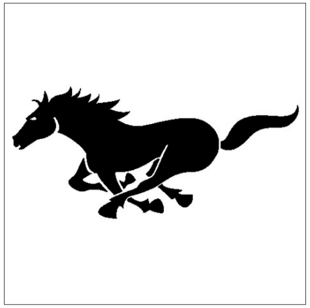
Moralité : plantez des arbres !

Structure **omniprésente** en informatique et en mathématiques appliquées

Certifié développement durable

TP : compression d'image



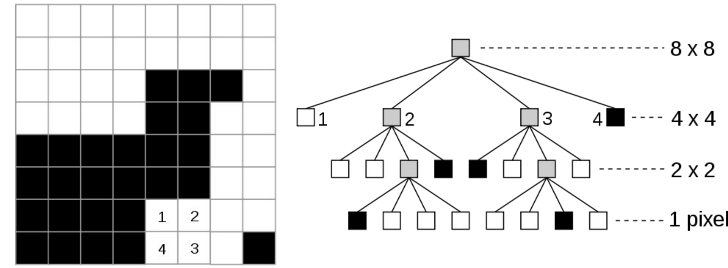


1) Récupérer la classe **QuadTree<T>**, lire l'exemple associé.
   La considérer comme une bibliothèque : ne pas modifier le fichier.
2) Considérer une image noir&blanc carrée de taille une puissance de 2 (ex. d'image 512x512 fournie) : la charger, l'afficher
3) Encoder l'image dans un **QuadTree<bool>** [≈ 30 LOC] :

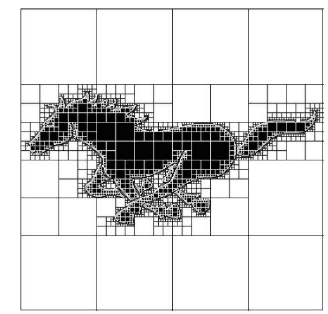
algorithme

- descendre récursivement dans les sous-régions carrées de l'image
- **attention :** ne pas créer de sous-images, manipuler juste les coordonnées des régions
- **attention :** ne chercher à stoker les coordonnées de chaque région du quadtree, elles sont déduites du carré initial, calculées au vol quand on descend dans l'arbre
- construire le quadtree « en remontant » de la récursion :
  - quand on atteint une région de taille 1x1 (= un pixel)
    on retourne une feuille de la couleur du pixel
  - quand on reçoit 4 quadtrees correspondant à l'encodage des 4 sous-régions
    s'ils ont la même couleur, c.-à-d. si ce sont 4 feuilles de même couleur
    on retourne une feuille de cette couleur (inutile de la construire : 1 des 4)
    sinon on retourne un nouveau nœud avec ces 4 quadtrees comme fils
- **attention :** ne pas tester explicitement l'uniformité de couleur de toute une région, chaque pixel ne doit être testé qu'un seule fois (pas $\log_2(512)$ fois)

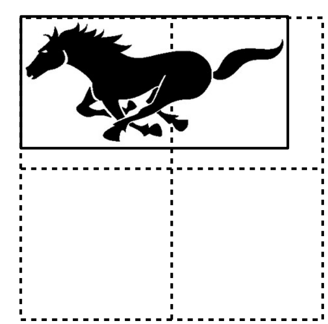
TP : compression d'image



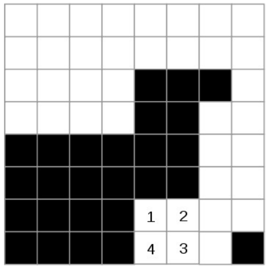
4) Optimiser : faire un *quaddag* qui partage toutes les feuilles Noir et Blanc
   - ne pas construire une nouvelle feuille blanche ou noire pour chaque pixel, n'en construire qu'une de chaque type avant l'encodage, dans des variables globales, et utiliser ces feuilles pré-construites lors de l'encodage d'un pixel.
   - utiliser la variable globale QuadTree<bool> ::**protect_leaves_from_destruction** pour ne pas libérer ces feuilles partagées dans le delete d'un QuadTree<bool>.
   - programmer dans la même fonction, cette variable indique si on souhaite un quaddag.
5) Décoder le quadtree en tant qu'image [≈ 25 LOC] :
   - créer une nouvelle image de même taille, parcourir récursivement le quadtree en remplissant au vol les quadrants de la nouvelle image
   - **attention :** pour une décompression efficace, remplir un tableau et ne l'afficher en tant qu'image qu'à la toute fin
   - [option] dessiner des carrés dans l'image pour voir les <u>grosses</u> feuilles

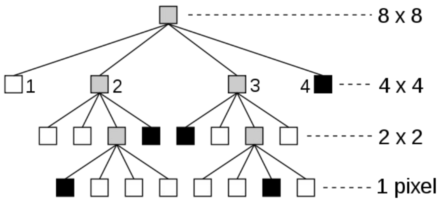


6) Compression :
   - estimer la taille de l'image compressée [taille des nœuds/feuilles du qtree via **<u>sizeof</u>**] distinguer QuadTree et QuadDAG car il n'y a que deux feuilles dans ce dernier cas.
   - comment mesurer <u>légitimement</u> le taux de compression ?
7) Traitement d'une image de dimension quelconque $\neq 2^N \times 2^N$ [≈ +10 LOC] :
   - arrondir les dimensions à la puissance de 2 supérieure
   - modifier les fonctions d'encodage et décodage.
   - **attention :** compléter <u>implicitement</u> avec du blanc hors de l'image (ou du noir...), ne pas créer explicitement une nouvelle image carrée



TP : compression d'image

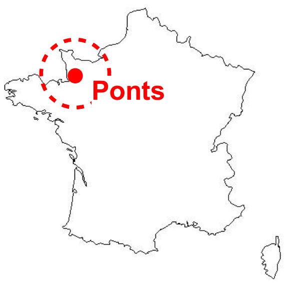




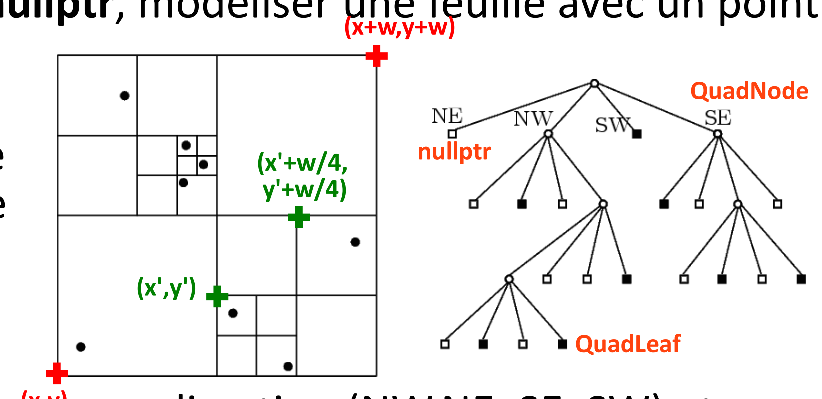
8) Traitement des niveaux de gris (compression avec perte, QuadTree<byte>) :
   - si 4 feuilles sont d'intensité voisine (différence max d'intensité < seuil), les remplacer par une seule feuille d'intensité moyenne
   - **attention :** le test doit être symétrique (sans favoriser une feuille)
   - variante pour éviter les dérives de dégradation : faire dépendre le seuil de la taille des régions (seuil ↘ quand région ↗)
   -
   -
   -

9) [Optionnel] Traitement de la couleur (QuadTree<Color>) :
   - faire comme pour les niveaux de gris, indépendamment sur chaque canal RVB

Exercice d'entraînement : quelle est la ville la plus proche des Ponts ?

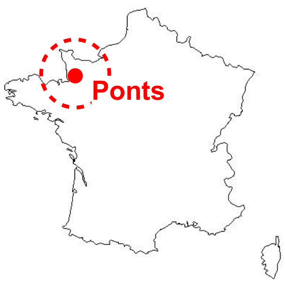


1. Récupérer les fichiers **quadtree.zip** et **villes1.zip** sur le site du cours. Lire les headers (.h).
   - **QuadTree<T>** : arbre à 4 branches avec un objet de type T aux feuilles
   - **Point2D<T>** : point dans $\mathbb{R}^2$ portant une information de type T
2. Coder le stockage de points 2D dans un quadtree, <u>pour un intervalle carré donné de $\mathbb{R}^2$</u> :
   - **indice 1 :** modéliser une feuille vide (sans point) par **nullptr**, modéliser une feuille avec un point 2D par un objet de type **QuadTree<Point2D<T>>***
   - **indice 2 :** les coordonnées des quadrants ne sont <u>pas stockées dans l'arbre</u>, elles sont déduites du carré initial, <u>calculées au vol</u> quand on descend dans l'arbre
   - **indice 3 :** considérer des carrés définis par un triplet (x,y,w), représentant les sommets (x,y)-(x+w,y+w). Voir pour ça au besoin le fichier **square.h**.



   - **indice 4 :** considérer les quadrants d'un carré, définis par une direction (NW,NE, SE, SW) et le sous-carré associé. Voir au besoin **quadrant.h**, implémenter **quadrant.cpp** [≈ 30 LOC]
   - **indice 5 :** voir l'interface de la fonction **insert** dans **neighbors.h** et la coder [≈ 40 LOC] cf. pp.64+
   - **attention :** si un point a les mêmes coordonnées qu'un point déjà dans l'arbre, il est à ignorer
   - **validation :** tester d'abord votre code avec qq points artificiels (affichage de l'arbre avec **display**)

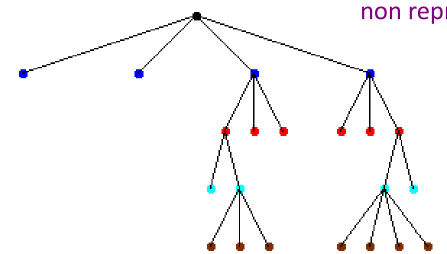
Exercice d’entraînement : quelle est la ville la plus proche des Ponts ?

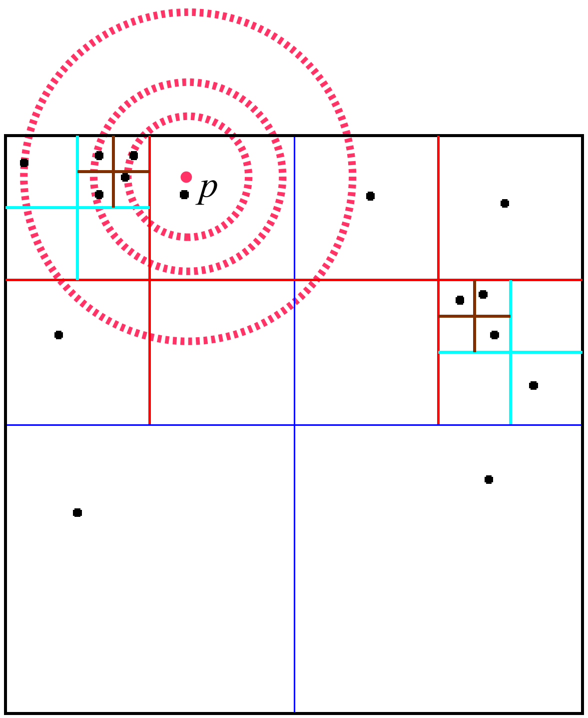


3. Implémenter, pour la distance euclidienne, la recherche de <u>tous les proches voisins</u> d'un point p donné, c.-à-d. à une distance $\leq r$ donné :
   - **algorithme :** descendre récursivement dans l'arbre (cf. p. 69)
   - **indice 1 :** implémenter une fonction **intersects_disk** qui teste si un carré donné intersecte un disque centré sur p de rayon r (cf. **square.h**) [≈ 20 LOC]
   - **astuce :** éviter sqrt, comparer des distances^2 (cf. p. 69)
   - **indice 2 :** voir l'interface de la première fonction **search** dans **neighbors.h** et la coder [≈ 40 LOC]
   - **validation :** vérifier sur un ensemble de quelques points

4. Ajouter la recherche du <u>plus proche voisin</u> :
   - **algorithme :** modification au vol de r (cf. p. 71+)
   - **attention :** <u>ne pas dupliquer de code</u>, ajouter un bool à votre fonction de recherche des proches voisins
   - **indice :** voir l'interface de la deuxième fonction **search** dans **neighbors.h** et la coder [≈ +5 LOC]
   - **validation :** vérifier sur un ensemble de quelques points

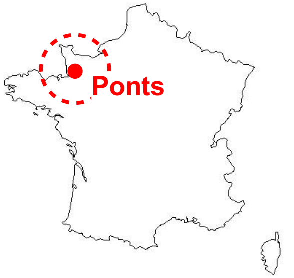
(branches à feuilles nulles non représentées)





© James Demmel

Exercice d'entraînement : quelle est la ville la plus proche des Ponts ?

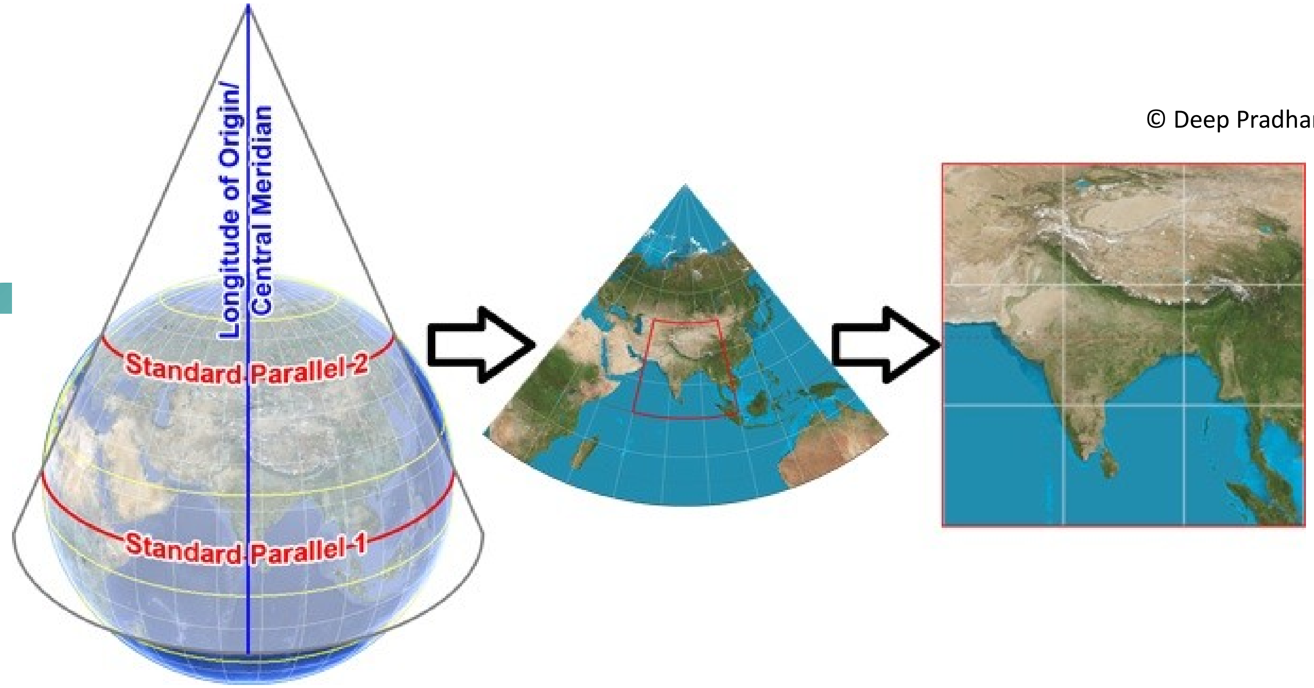


5. Charger les villes de France métropolitaine (**villes.txt**) dans **QuadTree<Point2D<Town>>* :**
   - lecture du fichier, dimensionnement du carré initial : voir **read_file** dans **town.h**, **town.cpp**
   - **attention :** les distances en latitudes-longitudes n'ont pas de sens, passer en Lambert93 (p. 89)
6. Quelle est la taille du quadtree ?
   - **indice :** cf. fonctions **nLeaves**, **nNodes**, **nTrees** de **QuadTree**
7. Quelle est la ville la plus proche de **Ponts** ?
   Combien de nœuds du quadtree faut-il parcourir pour trouver ça ?
   Comparer avec un parcours linéaire du vecteur de villes.

☺ Chercher l'adresse de la mairie de Ponts…

8. Quel est le temps moyen pour trouver une plus proche ville :
   (a) avec un quadtree ?
   (b) avec un vector ?
   - **indices :** tirer 100 villes au hasard dans le vecteur de villes (mesure du temps: voir **example.cpp**)
9. Combien de recherches de plus proches villes faut-il faire en moyenne pour rentabiliser le temps de construction du quadtree ?
10. [Optionnel] Prendre en compte le fait que certaines villes, du fait des arrondis, ont les mêmes coordonnées : les conserver toutes au lieu de ne garder que la première (cf. 2)

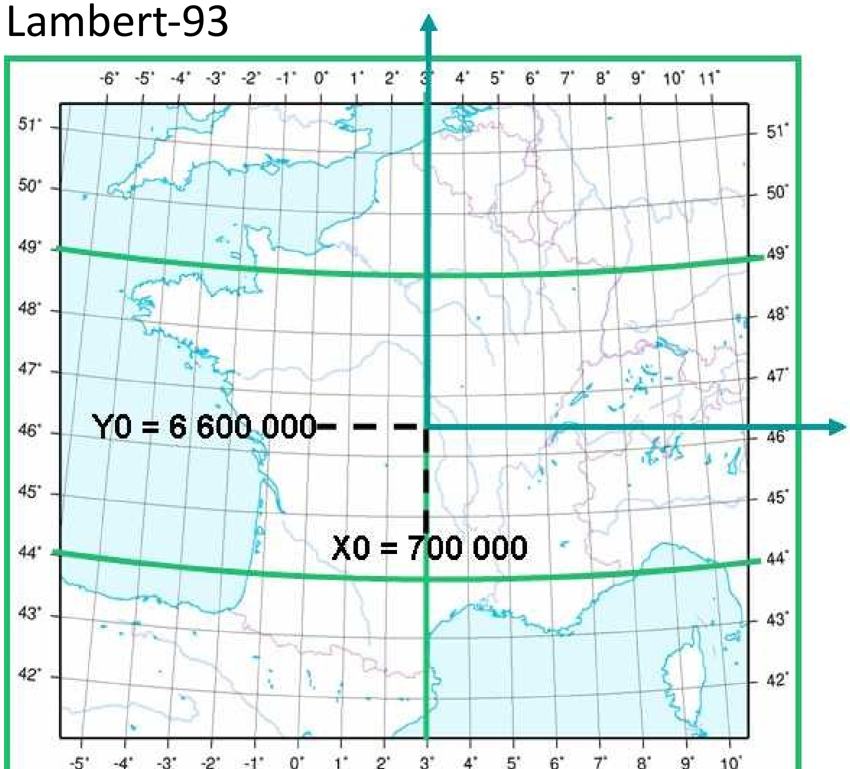
La carte et le territoire…

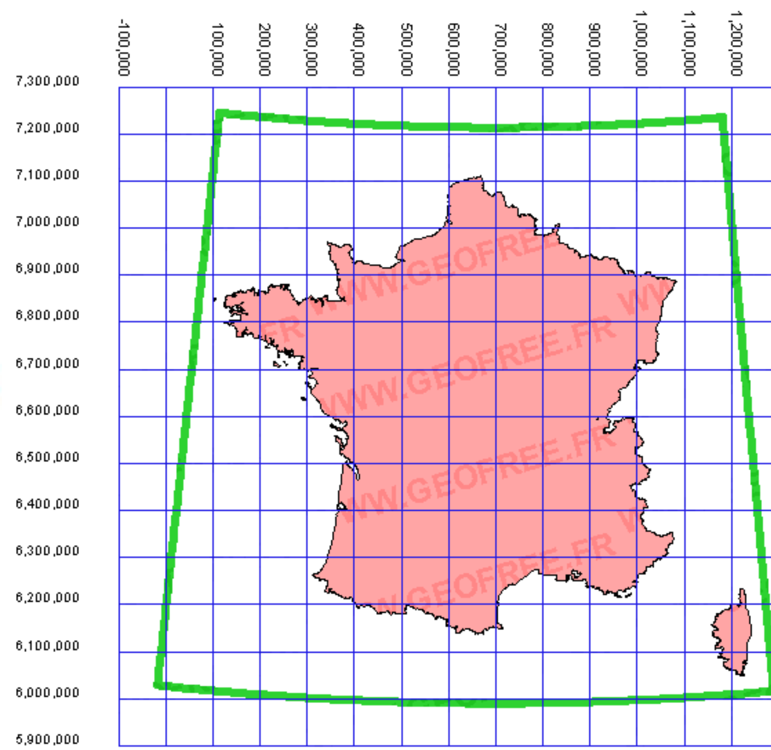


© Deep Pradhan

Projection conique conforme de Lambert

Lambert-93





© IGN

© geofree.fr

Bibliothèques

- Bibliothèque STL (structures de données courantes)
  - structures de données : vector, list, stack, queue, set...
  - http://www.cplusplus.com/reference/stl/
- Bibliothèque Imagine++ (images, graphisme)
  - site : http://imagine.enpc.fr/~monasse/Imagine++/
  - quick start :
    - http://imagine.enpc.fr/~monasse/Stereo/quickStartImagine++.pdf
    - http://imagine.enpc.fr/~monasse/Info/programming.pdf (annexe B)