

Algorithmique Avancée

Ecole Hassania des Travaux Publics
1^{ères} Année GI

Mohammed Karim Guennoun

Un algorithme, c'est quoi?

- Origine étymologique:
 - Vient du nom du mathématicien Al Khuwarizmi latinisé en « algoritmi »
- Définitions:
 - Académie Française: Méthode de calcul qui indique la démarche à suivre pour résoudre une série de problèmes équivalents en appliquant dans un ordre précis une suite finie de règles.
 - Wikipedia: Un algorithme est un moyen pour un humain de présenter la résolution par calcul d'un problème à une autre personne physique (un autre humain) ou virtuelle (un calculateur)

Algorithmique Vs programmation

- Un langage de programmation n'est que l'outil de réalisation de l'algorithme
- L'algorithme permet de concevoir la solution alors que la programmation permet de la mettre en œuvre
- L'algorithme est généralement à un niveau d'abstraction plus haut que la programmation
- Il est, parfois, nécessaire de prendre en compte la puissance et l'expressivité du langage de programmation avant de concevoir l'algorithme

La double problématique de l'algorithmique:

- Trouver **une méthode de résolution** (exacte ou approchée) du problème.
 - E.g. Soient trois nombres réels a , b et c , *quelles sont les solutions de l'équation ax^2+bx+c ?*
- Trouver **une méthode efficace**.
- Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre.



Motivation

Calcul de x^n

- **Données** : un entier naturel n et un réel x . On veut calculer x^n .
- **Moyens** :
 - Nous partons de $y_1 = x$.
 - Nous allons construire une suite de valeurs y_1, \dots, y_m telle que la valeur y_k soit obtenue par multiplication de deux puissances de x précédemment calculées :
 - $y_k = y_u * y_v$, avec $1 \leq u, v < k$, $k \in [2; m]$.
- **But** : $y_m = x^n$.
 - Le coût de l'algorithme sera alors de $m-1$, le nombre de multiplications faites pour obtenir le résultat recherché

Algorithme trivial

- $y = x$

Pour i de 2 à n faire

$y = y * x$

FinPour

renvoyer y

- *Le coût de l'algorithme: $n-1$ multiplications*

Méthode binaire

- Écrire n sous forme binaire
- Remplacer chaque :
 - 1 par la paire de lettres « SX » ;
 - 0 par la lettre « S ».
- Éliminer la paire « SX » la plus à gauche.
- Résultat : un mode de calcul de x^n , en partant de x où
 - S signifie « élever au carré » (*squaring*)
 - X signifie « multiplier par x ».
 - SX élevé au carré puis multiplier par X

Exemple

- Illustration avec $n = 25$
- $n = 11001$

1	1	0	0	1
SX	SX	S	S	SX
	SX	S	S	SX

- Nous partons de x et nous obtenons successivement :
- $x^2, x^3, x^6, x^{12}, x^{24}, x^{25}$.
- Nous sommes donc capables de calculer x^{25} en 6 multiplications au lieu de 24 pour l'algorithme trivial

La complexité

- Soit n un chiffre dont la représentation prend p chiffres binaires, on a:
 - $2^{p-1} \leq n < 2^p$
 - Ce qui donne: $p = \lfloor \log_2(n) \rfloor + 1$
- Soit $Nb1$, le nombre de 1 dans l'écriture binaire de n
 - On a donc : $(\lfloor \log_2(n) \rfloor + 1) - 1$ élévations au carré (élimination du premier SX)
 - Et $-1 + Nb1$ multiplications par x
- Le nombre de multiplications: $Nb1 + \lfloor \log_2(n) \rfloor - 1$

La complexité

- Trivialement :
 - $1 \leq Nb1 \leq |\log_2(n)|$
- On obtient donc:
 - $|\log_2(n)| \leq \text{Nbre multiplications} \leq 2|\log_2(n)|$
- Ce qui donne 20 multiplications au plus pour calculer x^{1000} au lieu de 999 multiplications

Y a t-il mieux?

- Prenons le cas $n = 15 = 1111$

1 1 1 1

SX SX SX SX

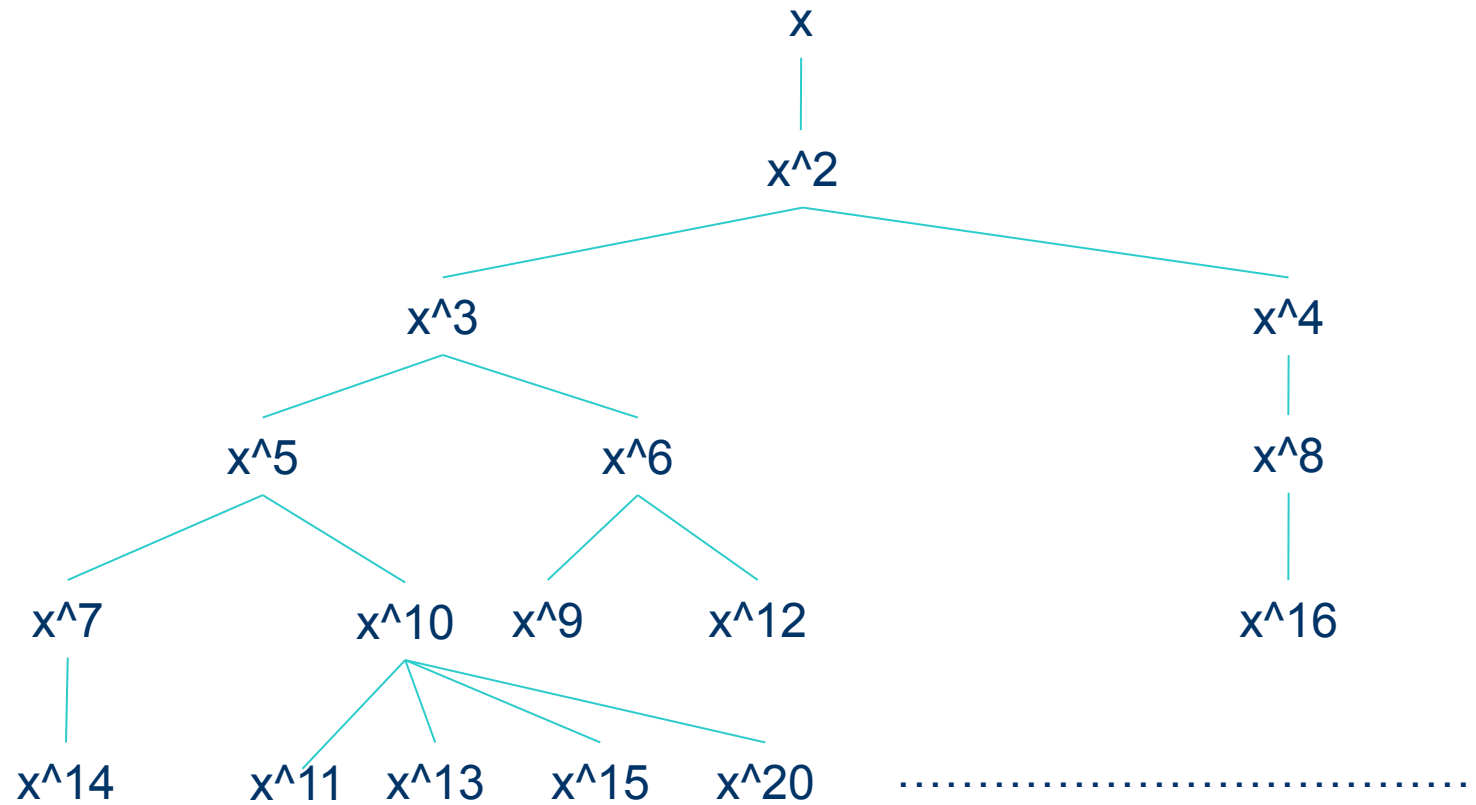
SX SX SX

- Nous partons de x et nous obtenons successivement :
 - $x^2, x^3, x^6, x^7, x^{14}, x^{15}$.
- Nous sommes donc capables de calculer x^{15} en **6 multiplications** par la méthode binaire
- Autre schéma de calcul : $x^2, x^3, x^6, x^{12}, x^{15} = x^{12} * x^3$.
- Nous obtenons ainsi x^{15} en **5 multiplications**
- La méthode binaire n'est donc pas optimale (c'est-à-dire que l'on peut faire mieux).

Algorithme de l'arbre

- Le $k+1$ e niveau de l'arbre est défini comme suit :
 - on suppose que l'on a déjà les k premiers niveaux ;
 - on construit le $k+1$ e de la gauche vers la droite en ajoutant sous le nœud n les nœuds de valeur $n+1, n+a_1, \dots, n+a_{k-1}$ où $1, a_1, \dots, a_{k-1}$ est le chemin de la racine au nœud n ;
 - on supprime tous les nœuds qui dupliquent une valeur déjà obtenue.

Illustration




Comparaison

- Pour calculer x^{15} , on effectue, donc le chemin:
 - $x^2 = x * x$
 - $x^3 = x^2 * x$
 - $x^5 = x^3 * x^2$
 - $x^{10} = x^5 * x^5$
 - $x^{15} = x^{10} * x^5$
- 5 multiplications au lieu de 6
- L'algorithme de l'arbre est optimal pour tout $n \leq 76$

En conclusion...

- Pour un problème donné (même aussi simple que calculer x^n), il est parfois facile de trouver un algorithme mais pas toujours l'algorithme le plus optimal
- La suite:
 - Concepts et définitions
 - Des problèmes classiques
 - Des méthodes classiques de résolution
 - Des structures de données classiques
 - Une étude de complexité comparative, sera réalisée tout au long



Complexité et optimalité: Concepts et définitions

La complexité

- Définition: *La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. La complexité est exprimée comme une fonction de la taille du jeu de données*
- *D_n , note l'ensemble des données de taille n*
- *$T(d)$, le coût de l'algorithme sur une donnée d*

Complexité au meilleur

- *Le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée.*
- *C'est une borne inférieure de la complexité de l'algorithme*
- $T_{min}(n) = \min_{d \in D_n} C(d)$

Complexité au pire

- C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée
- Il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué $T_{max}(n)+1$ opérations
- $T_{max}(n) = \max_{d \in D_n} C(d)$

Complexité en moyenne

- C'est la moyenne des complexités de l'algorithme sur les jeux de données d'une taille donnée
- Parfois, il faut prendre en compte la probabilité de leur apparition
- reflète le comportement « général » de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données

- $T_{moy}(n) = \frac{\sum_{d \in D_n} C(d)}{|D_n|}$

Optimalité

- *Un algorithme est dit optimal si sa complexité est la complexité minimale parmi les algorithmes de sa classe.*

Caractéristiques des études de complexité

- Dans une étude de complexité, on s'intéresse
 - Généralement à la *complexité en temps des algorithmes*.
 - *Parfois (rarement)* à d'autres caractéristiques
 - la *complexité en espace (taille de l'espace mémoire utilisé)*,
 - la largeur de bande passante requise,
 - ...

Contexte de l'étude

- Dans l'étude de complexité, il est parfois intéressant de considérer le contexte d'exécution
 - Accès aléatoire (RAM)
 - Processeur unique
 - Exécution séquentielle
 - ...

Ordre de grandeurs pour la complexité

- La complexité d'un algorithme est considérée d'un point de vue ordre de grandeur et pas en terme de complexité exacte.
- On utilise généralement les Notation de landau
 - $O : f = O(g) \leftrightarrow \exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c \times g(n)$
 - $o : f = o(g) \leftrightarrow \forall c \geq 0, \exists n_0, \forall n \geq n_0, f(n) \leq c \times g(n)$
 - $\theta : f = \theta(g) \leftrightarrow f = O(g) \text{ et } g = O(f)$



Illustration: Algorithme du tri par sélection

L'algo

- Idée très simple:
 - Parcourir le tableau,
 - Détecter l'élément le plus petit
 - Mettre cet élément à la position 0
 - Refaire le même traitement pour le deuxième élément le plus petit et l'indice 1
 - Continuer ainsi jusqu'à épuisement du tableau

Etude de complexité: tableau de taille n

- Pour** $j \leftarrow 0$ à $n-2$
 $\text{posmini} \leftarrow j$
 Pour $i \leftarrow j + 1$ à $n-1$
 Si $t(i) < t(\text{posmini})$ **Alors**
 $\text{posmini} \leftarrow i$
 Finsi
 i suivant
 $\text{temp} \leftarrow t(\text{posmini})$
 $t(\text{posmini}) \leftarrow t(j)$
 $t(j) \leftarrow \text{temp}$
 j suivant

c

n-1

c

$$\sum_{j=0}^{n-2} C_{\text{boucle}}(j)$$

c

n-1

c

n-1

c

n-1

- $C_{\text{boucle}}(j)$ = nbre d'exécutions de l'instruction pour un j donné

- Complexité = $4c(n-1) + \sum_{j=0}^{n-2} C_{\text{boucle}}(j)$

Complexité au meilleur cas

- Le meilleur cas se présente si le tableau est déjà trié
- Dans ce cas,
 - $C_{boucle}(j)=0$
 - La complexité est de:
 - $T(n)= 4c (n-1)$
- $T(n)$ est de la forme $an+b$
- C'est une complexité linéaire

Complexité au pire:

- La pire situation se présente quand le tableau est trié dans le sens inverse
- Dans ce cas
 - $C_{\text{boucle}}(j) = n - j - 1$
- On retrouve sachant que: $\sum_{j=1}^n j = \frac{n(n+1)}{2}$,
une complexité de la forme $an^2 + bn + c$
- Dans ce cas, nous avons une complexité quadratique.

Complexité en moyenne

- En moyenne, on peut avancer que pour un j donné, on va retrouver la moitié des valeurs plus grandes et l'autre plus petites
- Dans ce cas: $C_boucle(j) = \lfloor (n-1-j)/2 \rfloor$
- En remplaçant cette valeur dans l'équation, on retrouve aussi une complexité quadratique

Ordre de grandeur

- En se référant aux ordres de grandeur, on retrouve, pour l'algorithme de tri par sélection:
 - Complexité au mieux: $\theta(n)$
 - Complexité au pire: $\theta(n^2)$
 - Complexité en moyenne: $\theta(n^2)$

Autres classes des complexités

- Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité :
 - Les algorithmes sub-linéaires : $O(\log n)$
 - Les algorithmes linéaires: $O(n)$
 - *Les algorithmes quadratiques: $O(n^2)$*
 - Les algorithmes polynomiaux en $O(n^k)$ pour $k > 2$
 - Exponentiels: la complexité est supérieure à tout polynôme en n)
- Les algorithmes linéaires et sub-linéaires sont considérés comme rapides
- Les algorithmes polynomiaux sont considérés comme lents
- Les algorithmes exponentiels sont considérés comme très lents et impraticable au-delà d'une taille des données qui est supérieure à quelques dizaines d'unités.



Récurtivité et la méthode diviser pour régner

Le concept

- *Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir*
- *Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même*
- *Exemples typiques:*
 - *La factorielle*
 - X^n

Récurtivité simple

- On parle de récursivité simple lorsque l'algorithme contient un seul appel récursif
- Exemple:

$$x = \begin{cases} 1 & \text{si } n = 0 \\ x \times x^{n-1} & \text{si } n \geq 1 \end{cases}$$

Puissance (x,n):

Si n=0

Alors renvoyer (1)

Sinon renvoyer (x*Puissance(x,n-1))

FinSi

Récurtivité multiple

- On parle de récursivité multiple lorsque l'algorithme fait intervenir plusieurs appels récursifs
- Exemple:

$$C(n,p) = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon} \end{cases}$$

Si $p=0$ ou $p=n$

Alors renvoyer(1)

Sinon renvoyer($C(n-1,p)+C(n-1,p-1)$)

FinSi

Réversivité mutuelle

- On parle de réversivité mutuelle lorsque deux algorithmes s'appellent mutuellement de manière réversive

- Exemple:

$$\text{pair}(n) = \begin{cases} \text{vrai} & \text{si } n = 0 \\ \text{impair}(n-1) & \text{sinon} \end{cases}$$

pair(n)

Si n=0

alors renvoyer(vrai)

sinon renvoyer (impair(n-1))

$$\text{impair}(n) = \begin{cases} \text{faux} & \text{si } n = 0 \\ \text{pair}(n-1) & \text{sinon} \end{cases}$$

impair(n)

Si n=0

alors renvoyer(faux)

sinon renvoyer (pair(n-1))

Récurtivité imbriquée

- Exemple: la fonction d'Ackermann

$$A(m,n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m-1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m,n-1)) & \text{sinon} \end{cases}$$

Ackermann(m,n)

Si m=0

Alors renvoyer(n+1)

Sinon Si m>0 et n=0

Alors renvoyer (Ackermann(m-1,1))

Sinon renvoyer((Ackermann(m-1,Ackermann(m,n-1))))

Principe et intérêt

- Ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques
- On doit avoir :
 - Un moyen de se ramener d'un cas « compliqué » à un cas « plus simple »
 - Un certain nombre de cas dont la résolution est connue
 - Ces cas formeront les cas d'arrêt des traitements récursifs

Dangers / moyens

- Faire attention
 - À toujours retomber sur un cas connu, c'est-à-dire sur un cas d'arrêt
 - À ce que la fonction soit complètement définie, c'est-à-dire, qu'elle soit définie sur tout son domaine d'application
- Utiliser un ordre strict tel que
 - La suite des valeurs successives des arguments invoqués par la définition soit strictement monotone
 - Elle finira toujours par atteindre une valeur pour laquelle la solution est explicitement définie.

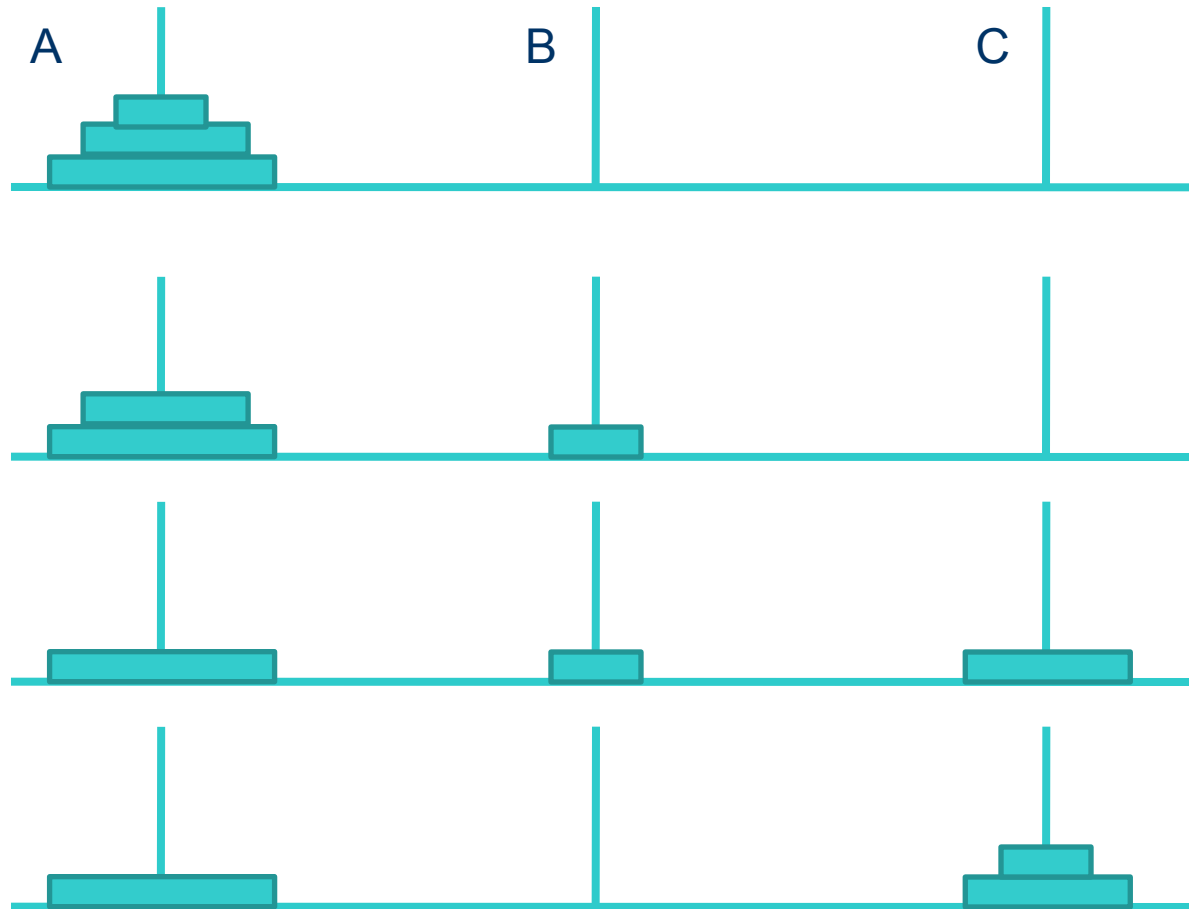


Exemple: les tours de
Hanoi

Le problème

- Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges.
- Sur ces tiges sont enfilés des disques de diamètres tous différents.
- Les seules règles du jeu sont que
 - l'on ne peut déplacer qu'un seul disque à la fois
 - il est interdit de poser un disque sur un disque plus petit.
- Au début tous les disques sont sur la tige de gauche, et à la fin sur celle de droite.

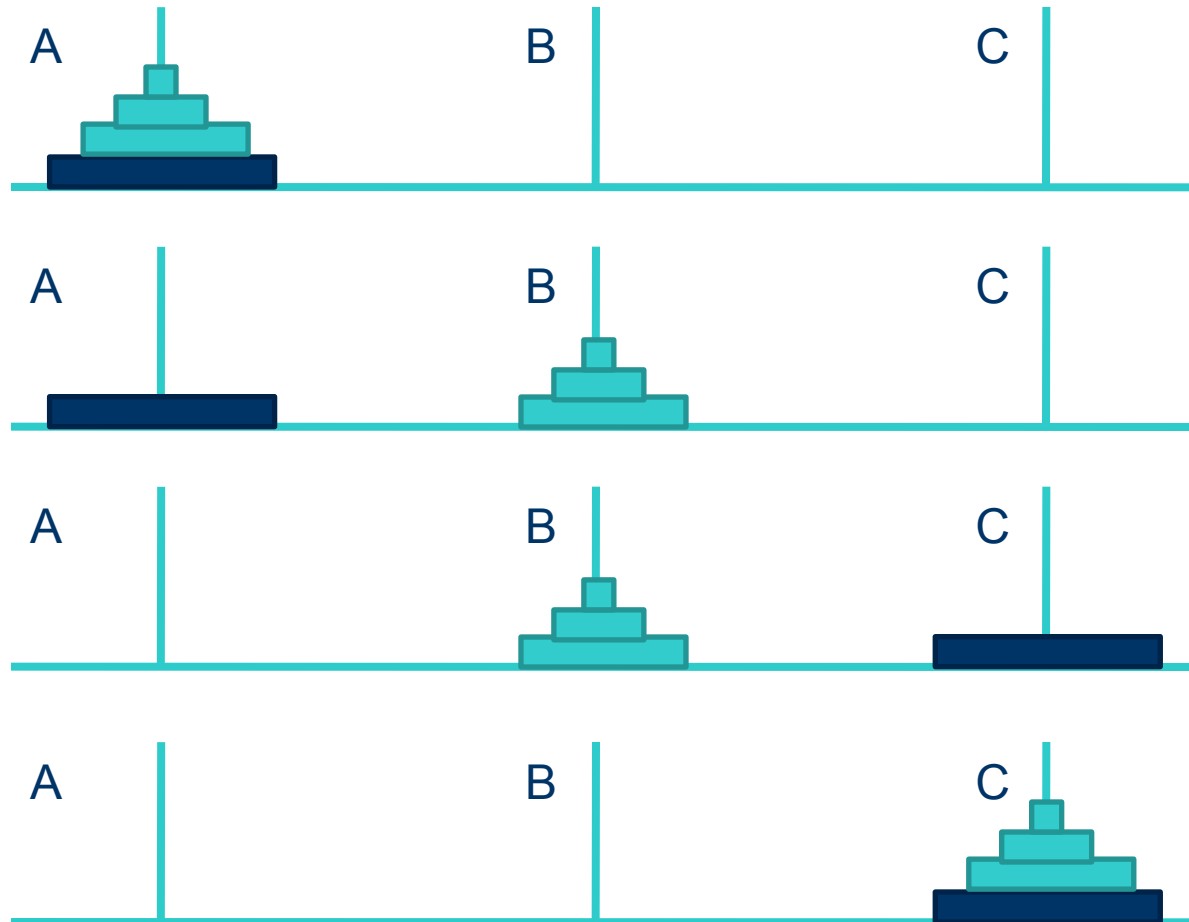
En schéma



Résolution

- Hypothèse : on suppose que l'on sait résoudre le problème pour $(n-1)$ disques.
- Principe : pour déplacer n disques de la tige A vers la tige C
 - on déplace les $(n-1)$ plus petits disques de la tige A vers la tige B,
 - Puis on déplace le plus gros disque de la tige A vers la tige C,
 - puis on déplace les $(n-1)$ plus petits disques de la tige B vers la tige C.

En schéma



L'algorithme

Hanoi(n,A,B,C)

Si $n=1$

Alors déplacerDisque(A,C)

Sinon

 Hanoi(n-1,A,C,B)

 déplacerDisque(A,C)

 Hanoi(n-1,B,A,C)

FinSi

Complexité de l'algo

- Soit n le nombre de disques, calculons $H(n)$ le nombre déplacements de disques

- On trouve:

$$H(n) = \begin{cases} 1 & \text{si } n = 1 \\ H(n-1) + 1 + H(n-1) & \text{sinon} \end{cases}$$

- Ce qui donne: $H(n) = 2 * H(n-1) + 1$ pour $n > 1$
- Et implique que: $H(n) = 2^n - 1$
- C'est une complexité exponentielle

La méthode diviser pour régner

- Principe:
 - Algorithmes avec une structure récursive
 - Appel récursif une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres,
 - Combinent les résultats pour trouver une solution au problème initial

Les étapes:

- 1. Diviser
 - Le problème est divisé en un certain nombre de sous-problèmes
- 2. Régner
 - Résoudre les sous-problèmes récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement
- 3. Combiner
 - Combiner les sous résultats pour produire le résultat final

Exemple multiplication de matrices

- Soient A et B deux matrices carrées de taille n
- *MULTIPLIER-MATRICES(A, B)*

Soit C une matrice carré de taille n

Pour i de 0 à n-1 faire

Pour j de 0 à n-1 faire

$C[i][j]=0$

Pour k de 0 à n-1 faire

*$C[i][j]= C[i][j] +A[i][k]*B[k][j]$*

FinPour

FinPour

FinPour

renvoyer C

- Complexité: $O(n^3)$ multiplications et additions

Un premier algo: diviser pour régner

- Supposons n est une puissance exacte de 2
- Décomposons les matrices A , B et C en sous-matrices de taille $(n/2) \times (n/2)$.
- L'équation $C = A * B$ peut alors se récrire :

$$\begin{bmatrix} r & s \\ t & u \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & g \\ f & h \end{bmatrix}$$

- Avec
 - $r = ae + bf$
 - $s = ag + bh$
 - $t = ce + df$
 - $u = cg + dh$

Complexité de l'algorithme

- L'addition de deux matrices de taille $n/2$ est de: $O(n^2)$
- On peut facilement prouver que:
 - $C(n) = 8C(n/2) + O(n^2)$

Résolution de la complexité pour la méthode diviser pour régner

- Théorème: Soient $a \geq 1$ et $b \geq 1$ deux constantes, soit $f(n)$ une fonction et soit $T(n)$ une fonction définie pour les entiers positifs par la récurrence : $T(n) = aT(n/b) + f(n)$
- $T(n)$ peut alors être bornée asymptotiquement comme suit :
 - Si $f(n) = O(n^{(\log_b a) - \epsilon})$ pour $\epsilon > 0$ alors $T(n) = \theta(n^{(\log_b a)})$
 - Si $f(n) = \theta(n^{(\log_b a)})$ alors $T(n) = \theta(n^{(\log_b a)} \log n)$

Application à notre algorithme

- On a:
 - $a=8$
 - $b=2$
 - $f(n) = \theta(n^2)$
 - $\log_b a = 3$
- On se retrouve dans le cas 1 avec $\epsilon=1$
- La complexité de l'algorithme est donc de n^3

Les algorithmes de tri





Tri par fusion

Principe (arrêt)

- L'algorithme de tri par fusion est construit suivant le paradigme « diviser pour régner » :
 - Il divise la séquence de n nombres à trier en deux sous-séquences de taille $n/2$.
 - Il trie récursivement les deux sous-séquences.
 - Il fusionne les deux sous-séquences triées pour produire la séquence complète triée.
- La récursion termine quand la sous-séquence à trier est de longueur 1 car une telle séquence est toujours triée.

L'algorithme

- Le principe central de l'algorithme se base sur l'étape de fusion de deux listes triées
 - à chaque étape, on compare les éléments minimaux des deux sous-listes triées, le plus petit des deux étant l'élément minimal de l'ensemble on le met de côté et on recommence.
 - On s'arrête à l'épuisement d'un des deux tableaux
 - A ce moment, il ne reste que la copie des élément du tableau qui n'est pas épuisé.

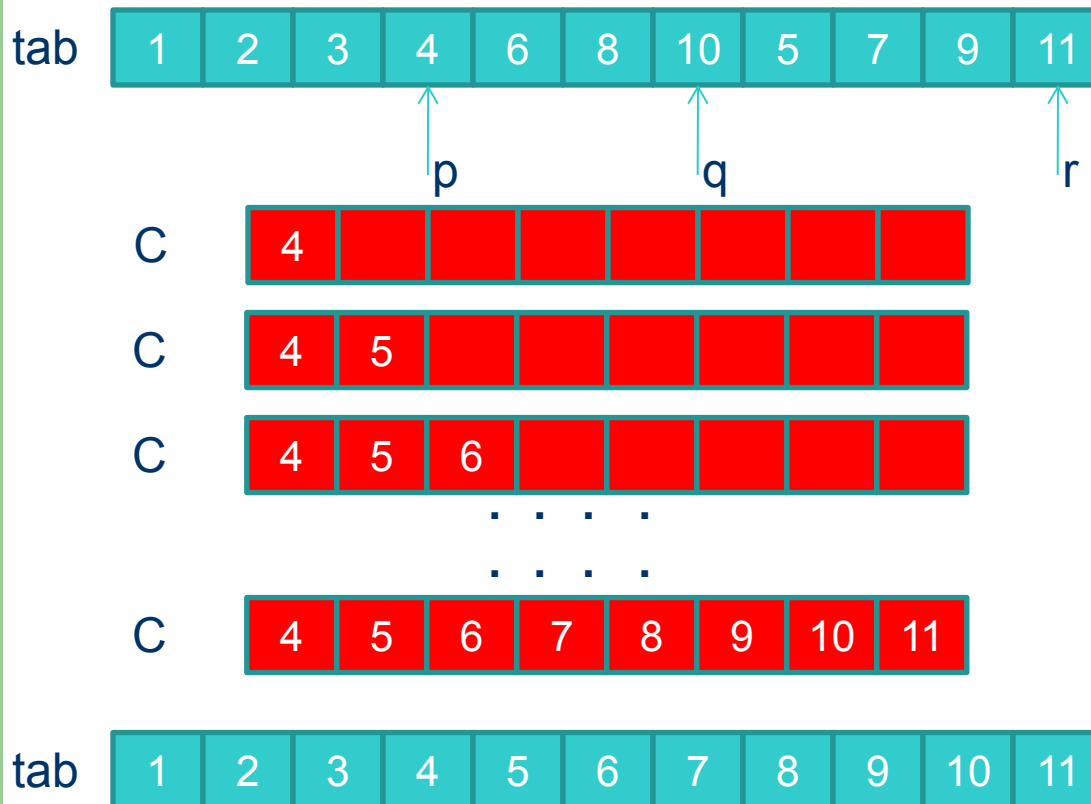
L'algorithme de fusionnement

FUSIONNER(A, p, q, r) // A le tableau initial, p le premier élément de la première partie
 // q son dernier élément, r le dernier élément de la deuxième partie

1. $i \leftarrow p$ // indice de parcours de la première partie
2. $j \leftarrow q+1$ // indice de parcours de la deuxième partie
3. Soit C un tableau de taille $r-p+1$ // tableau intermédiaire pour le calcul
4. $k \leftarrow 0$
5. Tant que $i \leq q$ et $j \leq r$ faire
6. Si $A[i] < A[j]$ alors $C[k] \leftarrow A[i]; i \leftarrow i+1;$
7. Sinon $C[k] \leftarrow A[j]; j \leftarrow j+1;$
8. FinSi
9. $k \leftarrow k+1$
10. FinTantQue
11. Tant que $i \leq q$ faire $C[k] \leftarrow A[i]; i \leftarrow i+1; k \leftarrow k+1; \text{FinTantQue}$
12. Tant que $j \leq r$ faire $C[k] \leftarrow A[j]; j \leftarrow j+1; k \leftarrow k+1; \text{FinTantQue}$
13. Pour $k \leftarrow 0$ à $r-p$ faire
14. $A[p+k] \leftarrow C[k]$

Déroulement de l'algorithme

- Fusionner(tab,4,7,11):



Complexité de l'algorithme de fusion

- Étudions les différentes étapes de l'algorithme :
 - les initialisations ont un coût constant $O(1)$;
 - la première boucle tant que de fusion s'exécute au plus $r-p$ fois, chacune de ses itérations étant de coût constant, d'où un coût total en $O(r-p)$;
 - les deux dernières boucles tant que complétant C ont une complexité respective au pire de $q-p+1$ et de $r-q$, ces deux complexités étant en $O(r-p)$;
 - la recopie finale coûte $O(r-p+1)$.
- Par conséquent, l'algorithme de fusion a une complexité en $O(r-p)$.

Algorithme de tri par fusion

TRI-FUSION(A, p, r)

Si $p < r$

Alors

$q \leftarrow \lfloor (p+r)/2 \rfloor$

TRI-FUSION(A, p, q)

TRI-FUSION($A, q+1, r$)

FUSIONNER(A, p, q, r)

FinSi

Complexité de l'algorithme de tri

- Complexité des trois phases de l'algorithme « diviser pour régner » :
 - **Diviser** : cette étape se réduit au calcul du milieu de l'intervalle $[p,r]$, sa complexité est donc en $O(1)$.
 - **Régner** : l'algorithme résout récursivement deux sous-problèmes de tailles respectives $n/2$ d'où une complexité en $2Comp(n/2)$.
 - **Combiner** : la complexité de cette étape (la fusion) est celle de l'algorithme de fusion qui est de $O(n)$ pour la construction d'un tableau solution de taille n .
- La complexité du tri par fusion est donnée par la récurrence
$$Comp(n) = \begin{cases} O(1) & \text{si } n = 1 \\ 2Comp\left(\frac{n}{2}\right) + O(n) & \text{sinon} \end{cases}$$

Déduction de la complexité

- En utilisant le théorème de résolution, on retrouve les valeurs $a=2$, $b=2$ et $f(n)=O(n)$, pour la formule $T(n) = aT(n/b) + f(n)$
- Sachant que $\log_2(2) = 1$, nous nous retrouvons dans le deuxième cas du théorème
- Ceci donne une complexité de :
 - $O(n \log(n))$
- C'est plus efficace que le tri par sélection: $O(n^2)$!



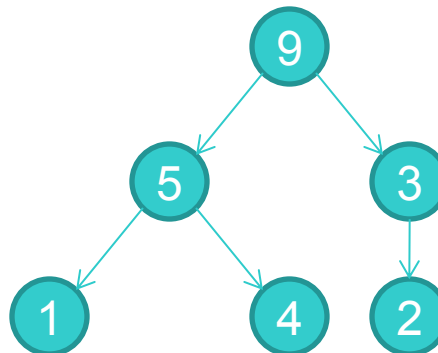
Tri par tas

C'est quoi un tas?

- Définition:

- *Un tas est un arbre binaire dont tous les niveaux sont complets sauf le dernier qui est rempli de la gauche vers la droite.*
- *Dans un tas, un père est toujours plus grand que ses deux fils.*

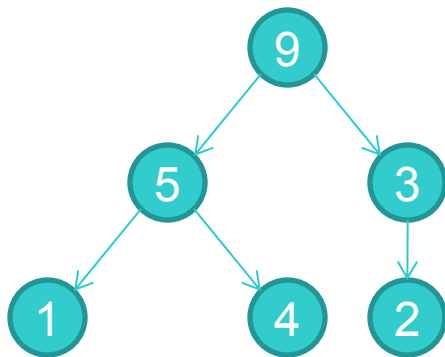
- Exemple



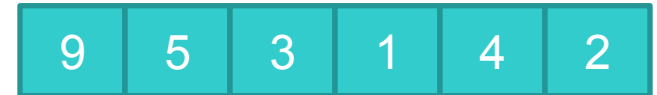
Représentation sous forme d'un tableau

- Un tas est représenté sous forme d'un tableau A telle que:
 - Le tableau A est un objet à deux attributs :
 - $\text{longueur}(A)$ qui est le nombre d'éléments qui peuvent être stockés dans le tableau A ;
 - $\text{taille}(A)$ qui est le nombre d'éléments stockés dans le tableau A .
 - La racine est stockée dans la première case du tableau $A[1]$.
 - Les éléments de l'arbre sont rangés dans l'ordre, niveau par niveau, et de gauche à droite.

Exemple de représentation



≡



Fonctions et propriété

- Les fonctions d'accès aux éléments d'un tableau A sont alors :
 - PÈRE(i): renvoyer $A(\lfloor i/2 \rfloor)$
 - FILS-GAUCHE(i): renvoyer $A(2*i)$
 - FILS-DROIT(i) renvoyer $A(2*i+1)$
- Propriété des tas :
 - $A[\text{PÈRE}(i)] \geq A[i]$

Problème de Conservation de la structure de tas

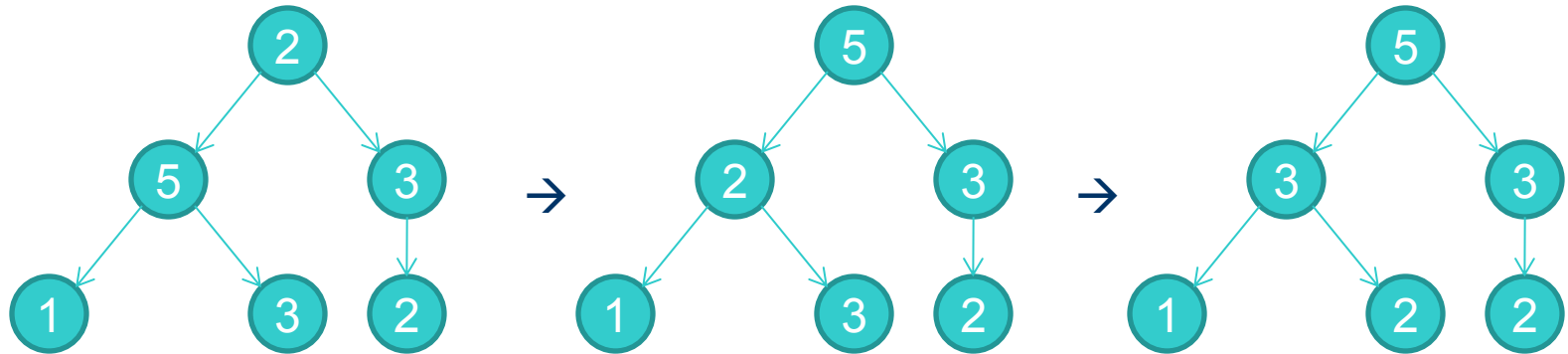
- Problème:
 - Étant donné un tableau A et un indice i .
 - En supposant que les sous arbres de racines $\text{Gauche}(i)$ et $\text{Droit}(i)$ constituent de tas
 - et que $A[i]$ viole la propriété des tas,
 - trouver un algorithme « Entasser » qui permette de réorganiser le tableau pour que le sous arbre de racine i soit un tas.

L'algorithme ENTASSER

ENTASSER(A, taille, i)

- $g \leftarrow 2*i$ // indice du fils gauche s'il existe
- $d \leftarrow (2*i)+1$ // indice du fils droit s'il existe
- $\text{max} \leftarrow i$
- si $g \leq \text{taille}(A)$ et $A[g] > A[\text{max}]$
alors $\text{max} \leftarrow g$
- si $d \leq \text{taille}(A)$ et $A[d] > A[\text{max}]$
alors $\text{max} \leftarrow d$
- si $\text{max} \neq i$
alors échanger $A[i]$ et $A[\text{max}]$;
ENTASSER($A, \text{taille}, \text{max}$)

Example



Complexité de l'algo ENTASSER (1/2)

- Le temps d'exécution de ENTASSER sur un arbre de taille n est en $O(1)$ plus le temps de l'exécution récursive de ENTASSER sur un des deux sous-arbres
- le pire cas survient quand la dernière rangée de l'arbre est exactement remplie à moitié.
- Le temps d'exécution de ENTASSER est donc décrit par

$$Comp(n) \leq Comp\left(\frac{2n}{3}\right) + O(1)$$

Complexité de l'algo ENTASSER(2/2)

- En utilisant le théorème de résolution, on retrouve les valeurs $a=1$, $b=3/2$ et $f(n)=O(1)$, pour la formule $T(n) = aT(n/b) + f(n)$
- Pour $\log_b(a) = 0$, nous nous retrouvons dans le deuxième cas du théorème
- Ceci donne une complexité de :
 - $O(\log(n))$

Construction d'un tas

- La construction se fait simplement par utilisation successive de l'algorithme ENTASSER
- *CONSTRUIRE-TAS(A, taille)*
 Pour $i \leftarrow \lfloor \text{taille}/2 \rfloor$ à 1
 ENTASSER(A, taille, i)
 i suivant

Illustration (1/2)

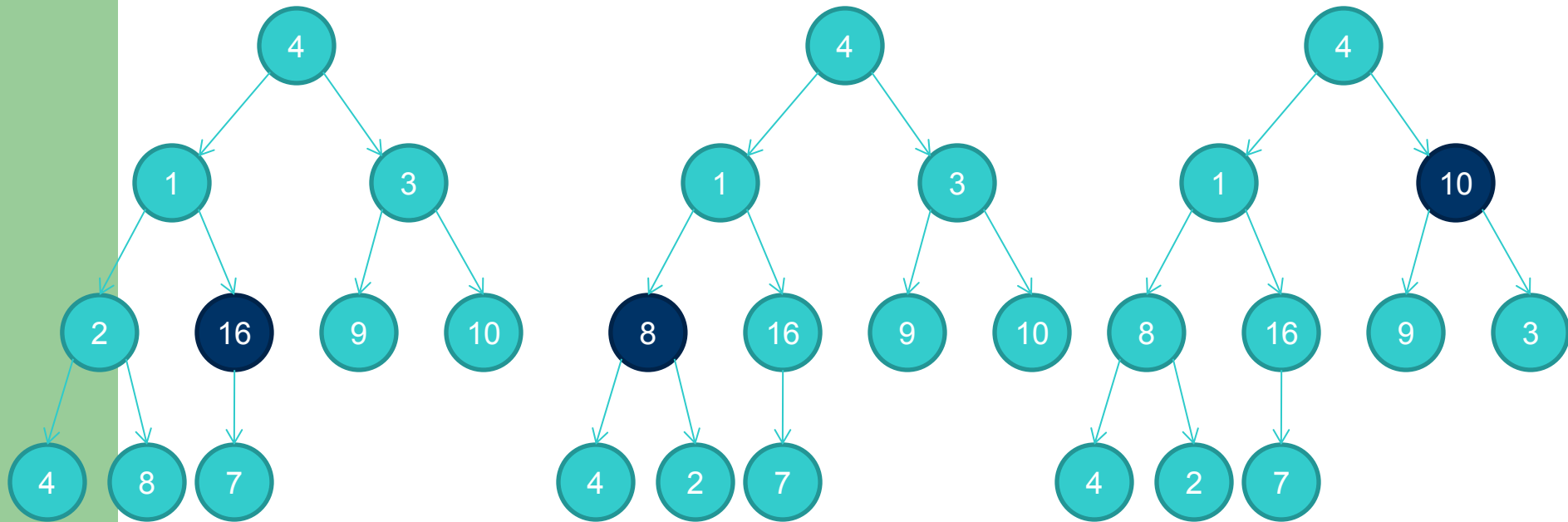
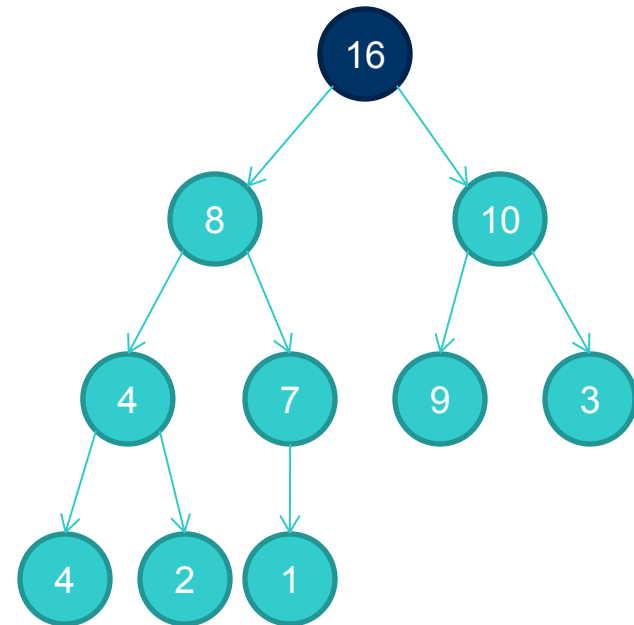
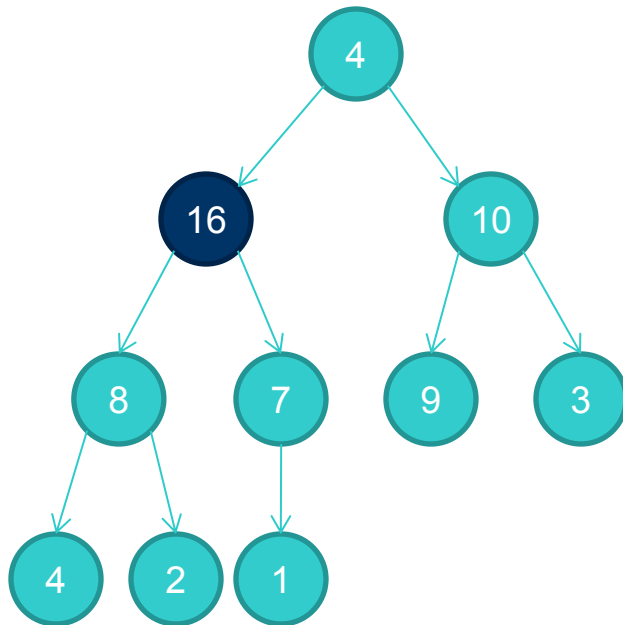


Illustration (2/2)



Complexité

- Borne supérieur:
 - Pour une construction d'un tas avec n valeurs,
 - $O(n)$ appels à Entasser (A, i)
 - Entasser(A, i) est de complexité $\log(i)$
 - Une borne supérieur pour la complexité est
 - $O(n \log(n))$

Une complexité plus fine pour construire tas

- un tas à n éléments
 - Considérons la hauteur inversée h (i.e. le niveau le plus bas correspond à $h=0$)
 - Une tas, possède une hauteur H de $\log_2(n)$
 - À une hauteur donnée h , il y a au maximum $\lfloor n/2^{h+1} \rfloor$ nœuds (réccursion)
 - L'algorithme ENTASSER requiert un temps d'exécution en $O(h)$ quand il est appelé sur un tas de hauteur h
 - On obtient, donc, en sommant sur la hauteur une complexité de:

$$Comp(n) = \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{n}{2^{h+1}} \right\rfloor O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \left\lfloor \frac{h}{2^h} \right\rfloor\right)$$

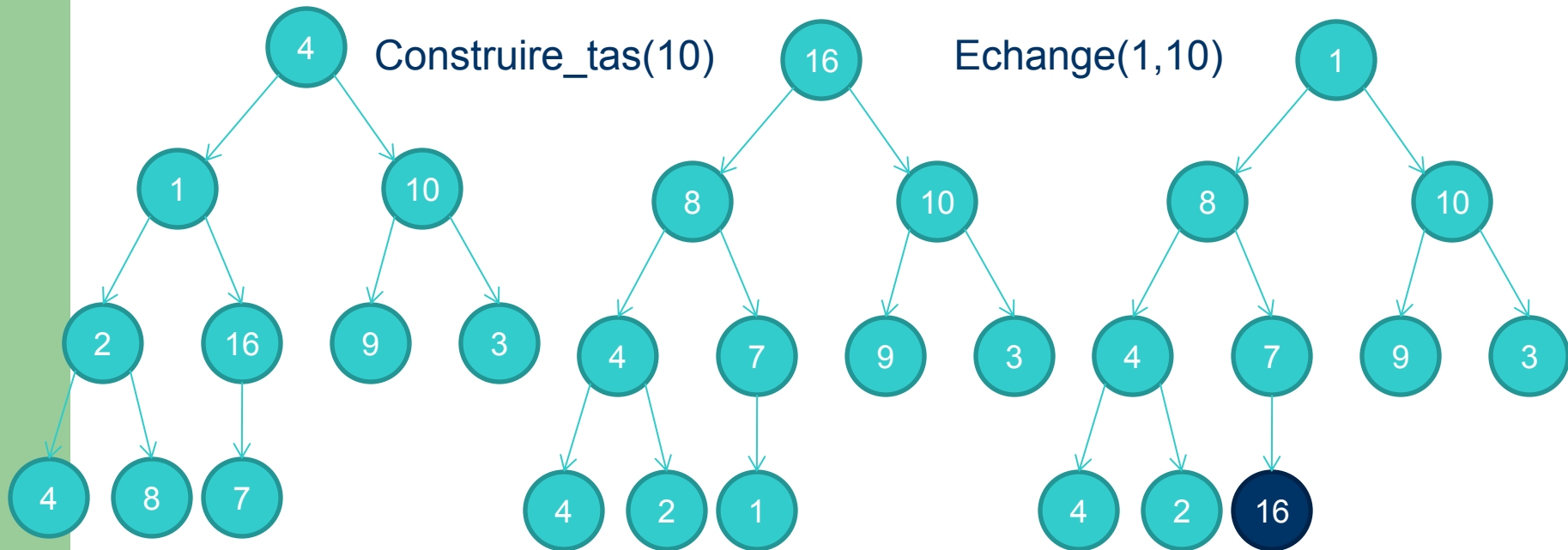
- Nous avons la série qui tend vers 2 en plus l'infini
- Nous obtenons une complexité de $O(n)$

Algorithme de tri par tas

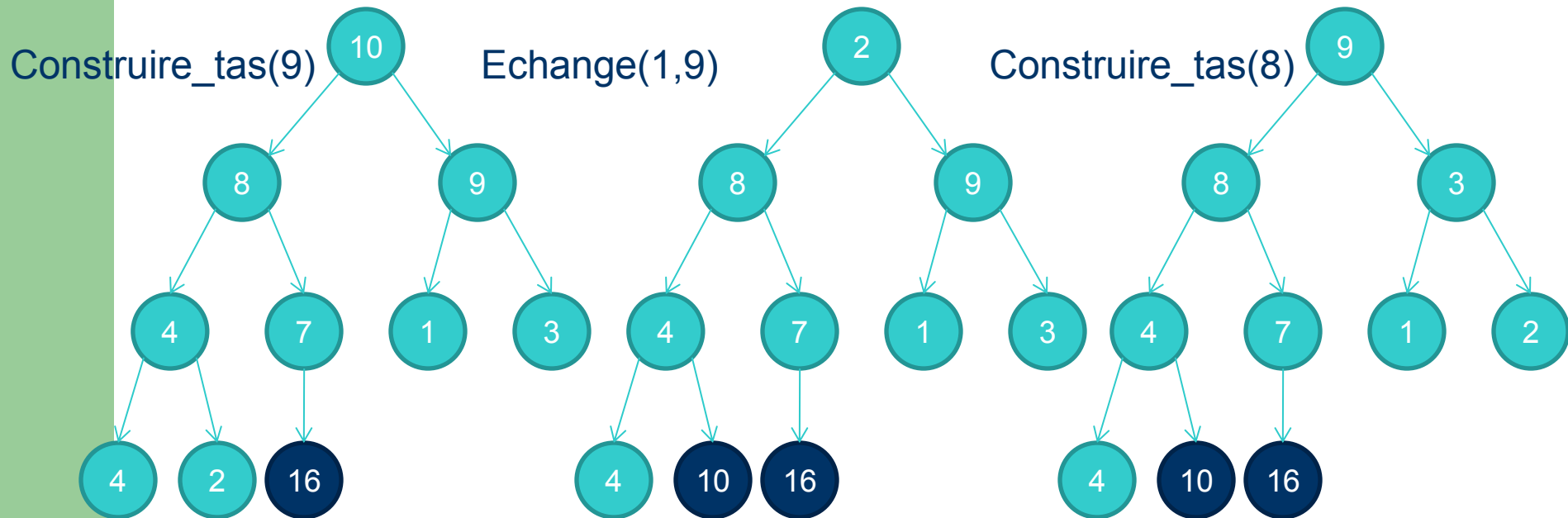
TRIER-TAS(A, taille)

- CONSTRUIRE-TAS(A, taille)
- **Pour $i \leftarrow \text{taille}$ à 2 faire**
- échanger $A[1]$ et $A[i]$
- $\text{taille} \leftarrow \text{taille} - 1$
- ENTASSER($A, \text{taille}, 1$)
- **i suivant**

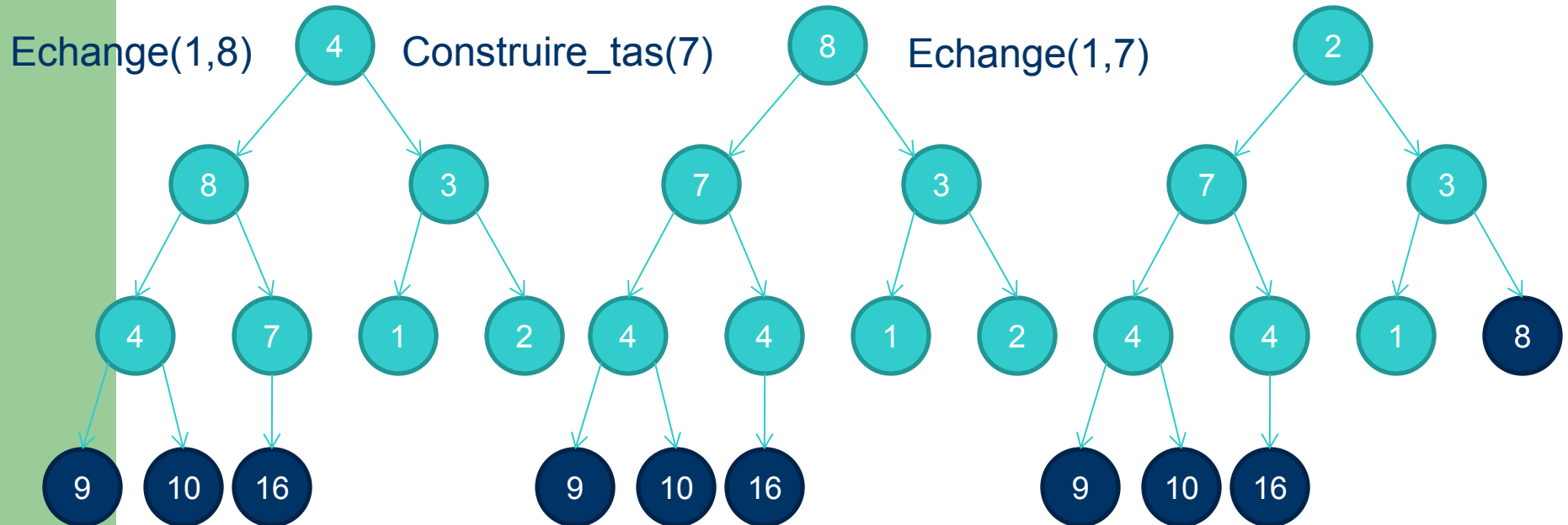
Illustration



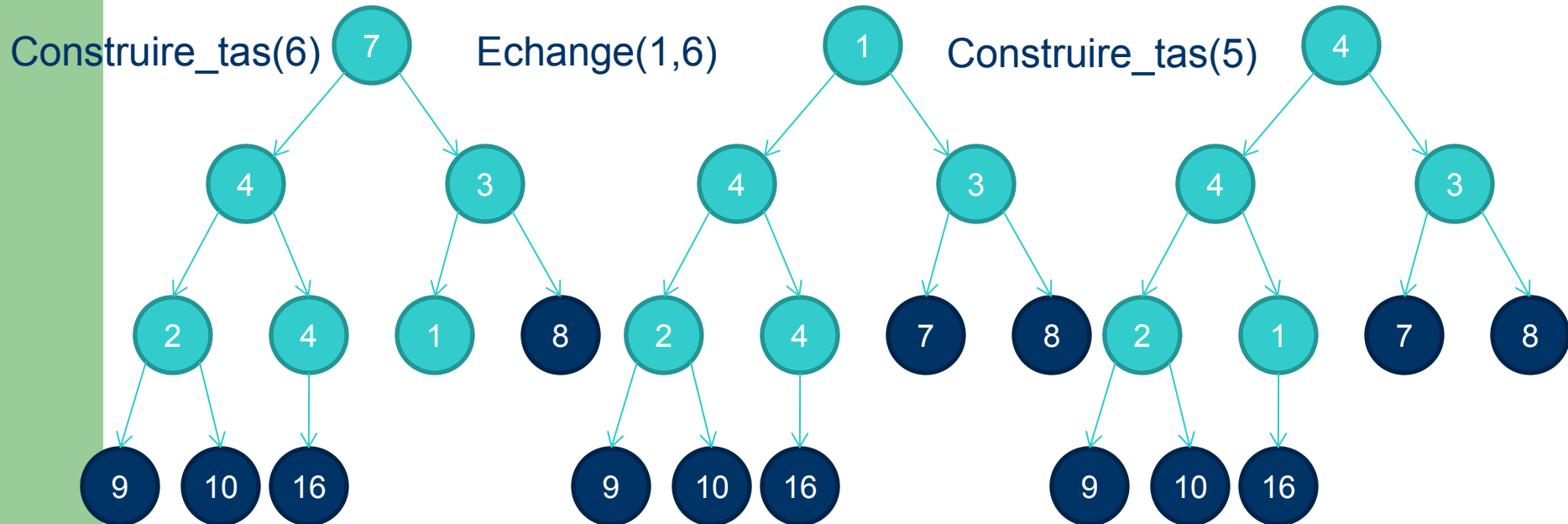
Illustration



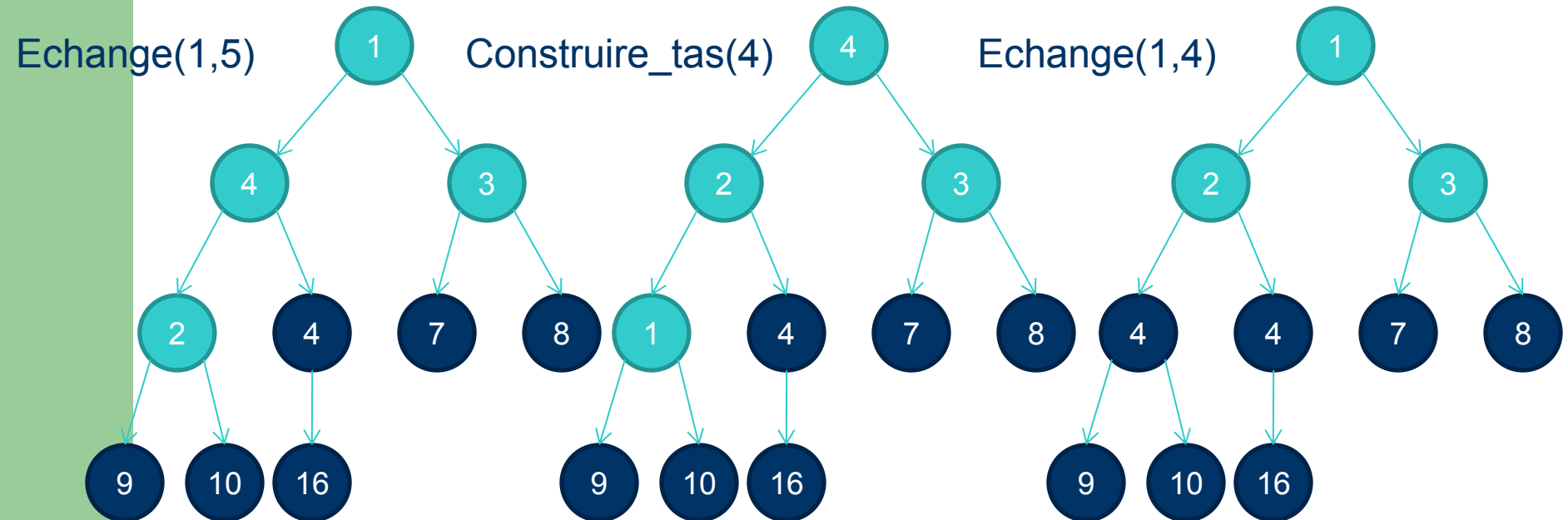
Illustration



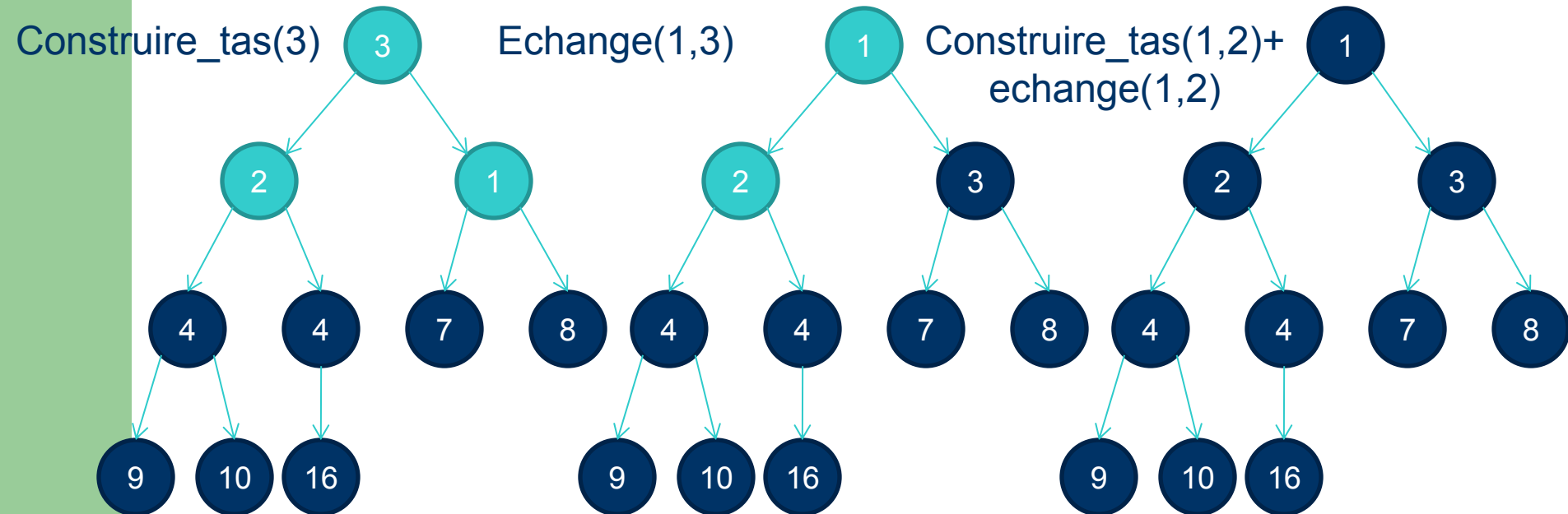
Illustration



Illustration



Illustration



Complexité de l'algorithme de tri par tas

- La procédure TRIER-TAS prend un temps $O(n \log_2(n))$ car l'appel à
- *CONSTRUIRE-TAS* prend un temps $O(n)$
- chacun des $n-1$ appels à *ENTASSER* prend un temps au maximum de $O(\log_2(n))$
- *TRIER_TAS* possède donc une complexité

$$O(n) + (n - 1) \times O(\log_2(n)) = O(n \log_2(n))$$



Structures de données élémentaires

Motivation

- Représentation des ensembles
- Un contenu rarement statique, besoin de le mettre à jour périodiquement
- Plusieurs possibilités de représentation
 - Aucune n'est meilleure dans l'absolu
 - En prenant en compte un besoin de traitement, une représentation peut être meilleure qu'une autre

Opérations élémentaires

- Généralement, ce type de structures, on définit les opérations suivantes:
 - **RECHERCHE(S, k)** : étant donné un ensemble S et une clé k, le résultat de cette requête est un pointeur sur un élément de S de clé k, s'il en existe un, et la valeur NIL sinon
 - **INSERTION(S, x)** : ajoute à l'ensemble S l'élément pointé par x.
 - **SUPPRESSION(S, x)** : supprime de l'ensemble S son élément pointé par x

Opérations élémentaires sur les ensemble ordonné

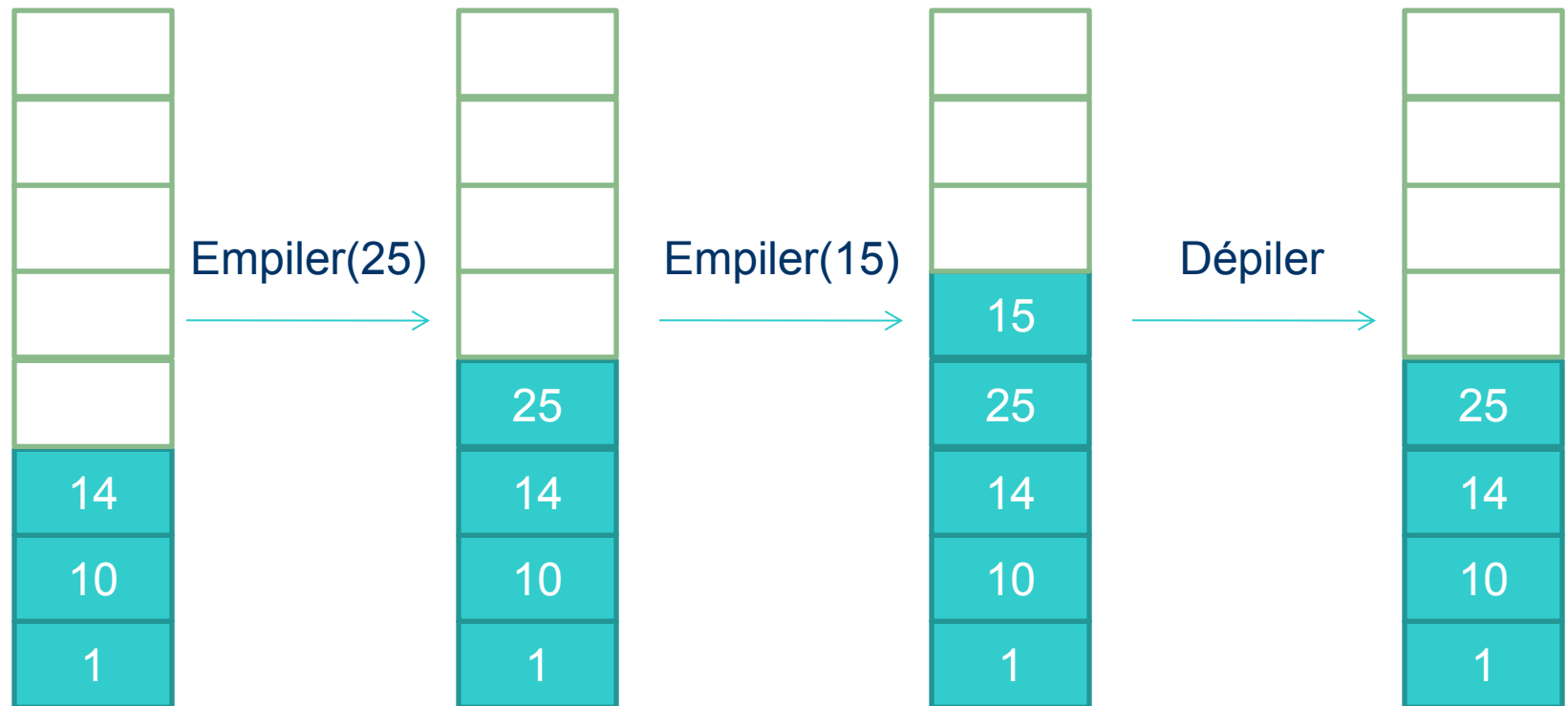
- Si l'ensemble est totalement ordonné, d'autres opérations sont possibles :
 - **MINIMUM(S)** : renvoie l'élément de S de clé minimale.
 - **MAXIMUM(S)** : renvoie l'élément de S de clé maximale.
 - **SUCCESSEUR(S, x)** : renvoie, si celui-ci existe, l'élément de S immédiatement plus grand que l'élément de S pointé par x , et NIL dans le cas contraire.
 - **PRÉDÉCESSEUR(S, x)** : renvoie, si celui-ci existe, l'élément de S immédiatement plus petit que l'élément de S pointé par x , et NIL dans le cas contraire.

Piles et files

Les piles

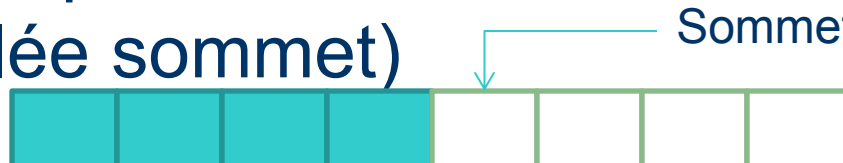
- *Une pile est une structure de données mettant en œuvre le principe*
 - *LIFO : Last-In, First-Out*
 - *dernier entré, premier sorti*
 - L'opération suppression est spécifiée par définition sur le dernier élément. Cette opération ne prend alors que l'ensemble comme argument.
 - L'opération insertion est aussi spécifiée par définition sur le dernier élément. Elle prend comme argument l'élément à insérer
 - Dans une pile, l'opération suppression est communément appelée dépiler et l'opération insertion est appelée empiler

Exemple



Utilisation de tableaux

- Une pile est facilement implémentée par un tableau
- Quelques points à prendre en compte lors de l'implémentation
 - Opération dépiler quand la pile est vide
 - Opération empiler quand la pile est pleine
- Représentation: un tableau avec une longueur, plus un pointeur sur la dernière case vide (appelée sommet)



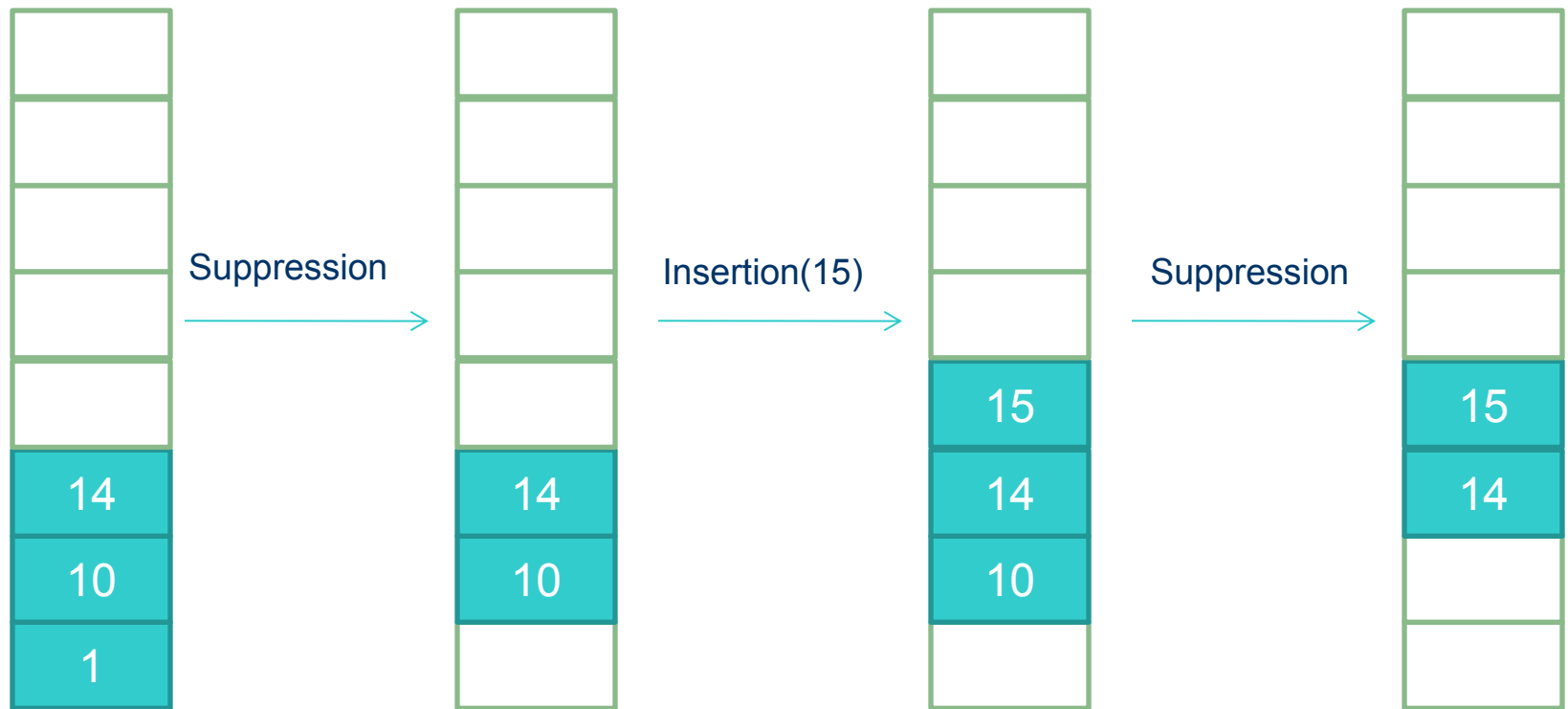
Algorithmes de traitement

- **PILE-VIDE(P)**
 - si $sommet(P)=0$ alors renvoyer *VRAI*
 - sinon renvoyer *FAUX*
- **EMPILER(P, x)**
 - si $sommet(P) = longueur(P)$ alors erreur « débordement positif »
 - sinon $P[sommet(P)] \leftarrow x$
 $sommet(P) \leftarrow sommet(P)+1;$
- **DÉPILER(P)**
 - si **PILE-VIDE(P)** alors erreur « débordement négatif »
 - sinon $sommet(P) \leftarrow sommet(P)-1$
 - renvoyer $P[sommet(P)]$

Les files

- *Une file est une structure de données mettant en œuvre le*
 - *FIFO : First-In, First-Out*
 - *premier entré, premier sorti*

Example



Utilisation de tableaux

- Une file est aussi facilement implémentée par un tableau
- Les mêmes cas de figure que les piles se présentent
 - Opération dépiler quand la pile est vide
 - Opération empiler quand la pile est pleine
- Représentation: un tableau plus deux pointeurs tête et queue et une variable booléenne plein



Les pointeurs tête et queue

- *tête(F): indexe (ou pointe) vers la tête de la file*
- *queue(F) qui indexe le premier élément libre de la file.*
- *PLEINE spécifie que la file est pleine (initialisé à faux)*
- *TAILLE donne la taille max de la file*
- *Les éléments de la file se trouvent donc aux emplacements $tête(F)$, $tête(F)+1$, ..., $queue(F)-1$ (modulo $TAILLE$).*

Algorithmes pour les files

- FILE-VIDE(F)
 - si ($tête(F)=queue(F)$ et non($PLEINE$)) alors renvoyer VRAI
 - sinon renvoyer FAUX
- INSERTION(F, x)
 - si $PLEINE$ alors erreur « débordement positif »
 - sinon $F[queue(F)] \leftarrow x$
 $queue(F) \leftarrow (queue(F)+1) \bmod(n)$
 $PLEINE \leftarrow (queue(F)==tête(F))$
- SUPPRESSION(F)
 - si FILE-VIDE(F) alors erreur « débordement négatif »
 - sinon $PLEINE \leftarrow FAUX$
 $ret \leftarrow F[tête(F)]$
 $tête(F) \leftarrow (tête(F)+1) \bmod(n)$
 $renvoyer (ret)$

Les listes chaînées

Définition

- Une liste chaînée est une structure de données dans laquelle les objets sont arrangés linéairement, l'ordre linéaire étant déterminé par des pointeurs sur les éléments
- Chaque élément de la liste, outre le champ clé, contient un champ successeur qui est pointeur sur l'élément suivant dans la liste chaînée.
- Si le champ successeur d'un élément vaut NIL, cet élément n'a pas de successeur et est donc le dernier élément de la liste chaînée

Manipulation de listes chaînées

- Le premier élément de la liste est appelé la tête de la liste
- Le dernier élément est appelé la queue de la liste
- Une liste L est manipulée via un pointeur vers son premier élément, que l'on notera $TÊTE(L)$
- *Si la liste est vide, $TÊTE(L)$ vaut NIL*

Illustration



↓ Suppression(5)



↓ Insertion(12)



Types des listes chaînées

- Simplement chaînée: liste comprenant le seul pointeur successeur
- doublement chaînée :
 - en plus du champ *successeur*, chaque élément contient un champ *prédécesseur* qui est un pointeur sur l'élément précédant dans la liste.
 - Si le champ *prédécesseur* d'un élément vaut *NIL*, cet élément n'a pas de prédécesseur et est donc la tête de la liste.

Illustration: les listes doublement chaînées



↓ Suppression(5)



↓ Insertion(12)



Types de listes chaînées

- Triée ou non triée : suivant que l'ordre linéaire des éléments dans la liste correspond ou non à l'ordre linéaire du contenu de ces éléments.
- Circulaire : si le champ prédécesseur de la tête de la liste pointe sur la queue, et si le champ successeur de la queue pointe sur la tête. La liste est alors vue comme un anneau.

Algorithme de recherche

- L'algorithme RECHERCHE-LISTE(L, k) :
 - trouve le premier élément de clé k dans la liste L par une simple recherche linéaire, et retourne un pointeur sur cet élément.
 - Si la liste ne contient aucun objet de clé k, l'algorithme renvoie NIL.
- RECHERCHE-LISTE(L, k)
 - $x \leftarrow \text{TÊTE}(L)$
 - tant que $x \neq \text{NIL}$ et $\text{clé}(x) \neq k$ faire
 - $x \leftarrow \text{successeur}(x)$
 - FinTantQue
 - renvoyer x
- L'algorithme marche pour les listes simplement et doublement chaînées

Algorithme d'insertion

- Étant donné un élément x et une liste L , l'algorithme INSERTION-LISTE insère x en tête de L .
- INSERTION-LISTE(L, x)
 - successeur(x) \leftarrow TÊTE(L)
 - **prédécesseur(x) \leftarrow NIL**
 - **Si TÊTE(L) \neq NIL Alors prédécesseur(TÊTE(L)) $\leftarrow x$**
 - **FinSi**
 - TÊTE(L) $\leftarrow x$
- L'algorithme traite les listes doublement chaînées
- Dans le cas des listes simplement chaînées, supprimer les instructions en gras

Algorithme de suppression (doublement chaînées)

- L'algorithme SUPPRESSION-LISTE:
 - élimine un élément x d'une liste chaînée L .
 - Cet algorithme a besoin d'un pointeur sur l'élément x à supprimer. Si on ne possède que la clé de cet élément, il faut préalablement utiliser l'algorithme RECHERCHE-LISTE pour obtenir le pointeur nécessaire.
- SUPPRESSION-LISTE(L, x)
 - Si $\text{prédécesseur}(x) \neq \text{NIL}$
Alors $\text{successeur}(\text{prédécesseur}(x)) \leftarrow \text{successeur}(x)$
Sinon $\text{TÊTE}(L) \leftarrow \text{successeur}(x)$ FinSi
 - Si $\text{successeur}(x) \neq \text{NIL}$
Alors $\text{prédécesseur}(\text{successeur}(x)) \leftarrow \text{prédécesseur}(x)$ FinSi

Algorithme de suppression (simplement chaînées)

- L'algorithme pour les listes simplement chaînées est plus complexe
 - Nous n'avons pas de moyen simple de récupérer un pointeur sur l'élément qui précède celui à supprimer
- SUPPRESSION-LISTE(L, x)
 - Si $x = \text{TÊTE}(L)$
 - Alors $\text{TÊTE}(L) \leftarrow \text{successeur}(x)$
 - Sinon $y \leftarrow \text{TÊTE}(L)$
 - TantQue $\text{successeur}(y) \neq x$ faire $y \leftarrow \text{successeur}(y)$
 - FinTantQue
 - $\text{successeur}(y) \leftarrow \text{successeur}(x)$
 - FinSi

100%

- Rechercher (L,K), dans le pire cas
 - Pour les listes:
 - $O(n)$: il faut parcourir toute la liste des successeurs pour récupérer l'élément
 - Pour les tableaux
 - $O(n)$: idem

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau trié	Tableau non trié
Rechercher(L,k)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)

Comparaison de complexité: l'opération insertion

- Insertion (L,x), dans le pire cas
 - Pour les listes non triées:
 - $O(1)$: il suffit de l'insérer au début
 - Pour les listes triées
 - $O(n)$, pire cas quand il faut l'insérer en dernier
 - Pour les tableaux non triés
 - $O(n)$: redimensionnement et création d'un nouveau tableau de taille supérieure
 - Pour les tableaux triés
 - $O(n)$: en plus il faut décaler les suivants (pire cas, un élément k plus petit que tous les éléments du tableau)

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau trié	Tableau non trié
Insertion(L,x)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$

Comparaison de complexité: l'opération suppression

- Suppression (L,x), dans le pire cas
 - Pour les listes simples:
 - $O(n)$: il faut chercher le prédécesseur
 - Pour les listes doubles
 - $O(1)$
 - Pour les tableaux
 - $O(n)$: il faut décaler les suivants (pire cas, supprimer le premier élément)

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau trié	Tableau non trié
Suppression(L,x)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Comparaison de complexité: l'opération successeur

- Successeur (L,x), dans le pire cas
 - Pour les listes ou les tableaux non triés:
 - $O(n)$: il faut parcourir toute la liste si c'est le dernier
 - Pour les listes ou les tableaux triés
 - $O(1)$: c'est l'élément d'indice $i+1$ pour les tableaux et c'est le successeur pour les listes

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Successeur(L,x)	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$

Comparaison de complexité: l'opération prédécesseur

- Prédécesseur (L,x), dans le pire cas
 - Pour les listes non triées
 - $O(n)$: il faut parcourir toute la liste
 - Pour les listes simples triée
 - $O(n)$: il faut parcourir la liste pour trouver le prédécesseur
 - Pour les listes doubles triées
 - $O(1)$
 - Pour les tableaux non triés
 - $O(n)$: parcourir tout le tableau
 - Pour les tableaux triés
 - $O(1)$, c'est l'indice $i-1$

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Prédécesseur(L,x)	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$

Comparaison de complexité: l'opération minimum

- Minimum (L), dans le pire cas
 - Pour les listes et les tableaux non triés
 - $O(n)$: il faut parcourir toute la liste
 - Pour les listes et les tableaux triés
 - $O(1)$: C'est la tête pour les listes et l'indice 0 pour les tableaux

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Minimum(L)	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$

100%

- Maximum (L), dans le pire cas
 - Pour les listes et les tableaux non triés:
 - $O(n)$: il faut parcourir toute la liste si c'est le dernier
 - Pour les listes triés
 - $O(n)$: aller jusqu'à la fin de la liste
 - Pour les tableaux triés
 - $O(1)$: c'est l'élément à l'indice $n-1$

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Maximum(L)	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)

Tableaux de comparaison global

	Liste simple non triée	Liste simple triée	Liste double non triée	Liste double triée	Tableau non trié	Tableau trié
Rechercher(L,k)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
Insertion(L,x)	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)
Suppression(L,x)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Successeur(L,x)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)
Prédécesseur(L,x)	O(n)	O(n)	O(n)	O(1)	O(n)	O(1)
Minimum(L)	O(n)	O(1)	O(n)	O(1)	O(n)	O(1)
Maximum(L)	O(n)	O(n)	O(n)	O(n)	O(n)	O(1)

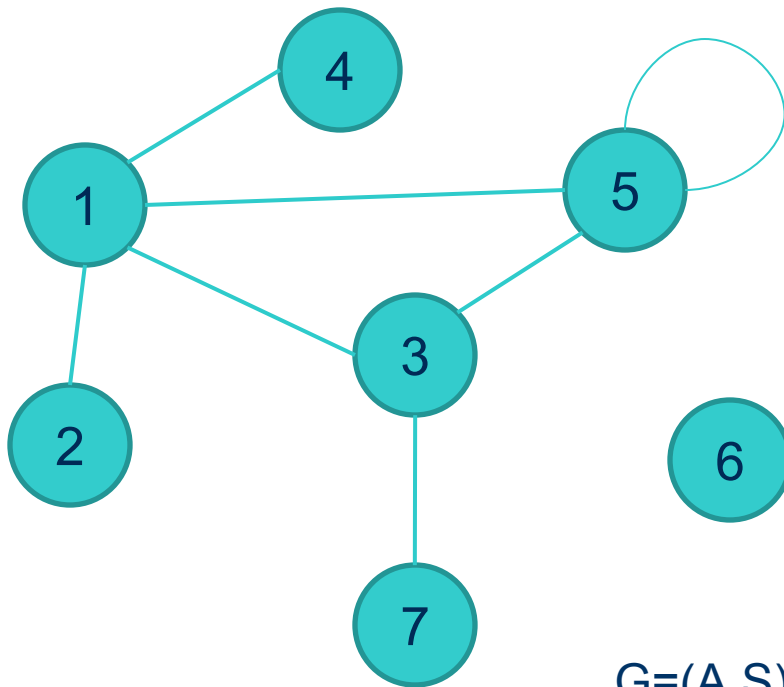
Graphes et arbres

Graphes

Graphes non orientés

- Un **graphe non orienté** G est représenté par un couple (S, A) .
 - S est un ensemble fini. Il correspond à l'ensemble des **sommets** (ou **nœuds**) de G
 - A une relation binaire sur S . Elle définit l'ensemble des **arrêtes** de G
- Graphiquement, les nœuds sont représentés par des cercles et les arcs par des traits
- Les boucles sont les arcs qui relient un nœud à lui-même

Example



$G=(A,S)$

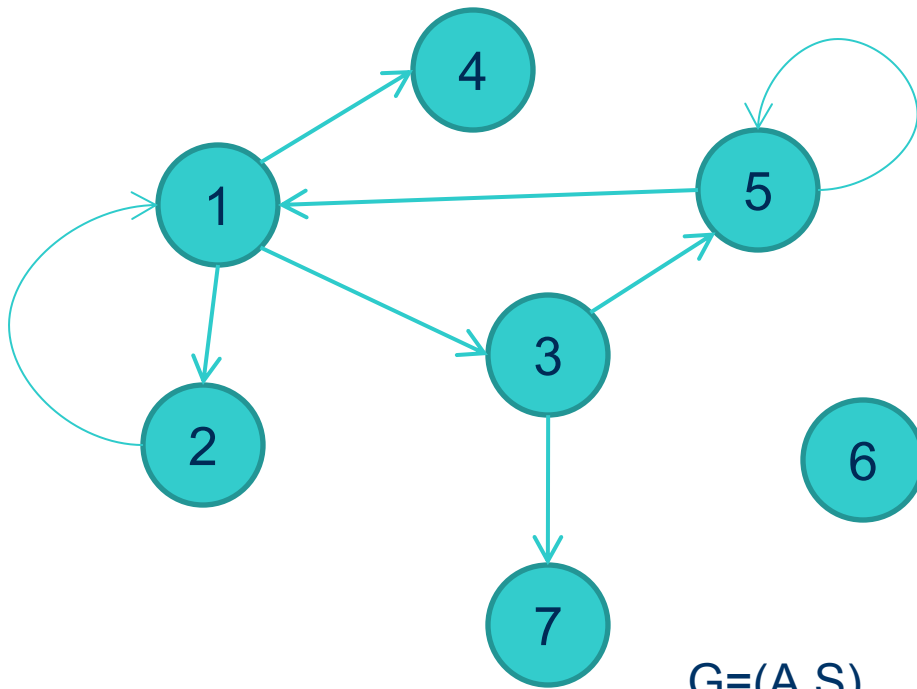
$A=\{1;2;3;4;5;6;7\}$

$S=\{\{1,2\};\{1,3\};\{1,4\};\{1,5\};\{3,5\};\{3,7\};\{5,5\}\}$

Graphes orientés

- Un **graphe orienté** G est représenté par un couple (S, A) .
 - S est un ensemble fini. Il correspond à l'ensemble des sommets (ou nœuds) de G
 - A une relation binaire sur S . Elle définit l'ensemble des **arcs** de G
- Graphiquement, les nœuds sont représentés par des cercles et les arcs par des flèches
- Les arcs sont définis par des couples (ordonnés) au lieu d'ensembles de deux éléments.

Example



$G=(A,S)$

$A=\{1;2;3;4;5;6;7\}$

$S=\{(1,2);(1,3);(1,4);(2,1);(3,5);(3,7);(5,1);(5,5)\}$

Lexique (graphes non orientés)

- Si $(u;v)$ est une arête d'un graphe non orienté $G = (S;A)$, on dit que l'arête $\{u,v\}$ est **incidente** aux sommets u et v .
- Dans un graphe non orienté, le **degré d'un sommet** est le nombre d'arêtes qui lui sont incidentes.
- Si un sommet est de degré 0, il est dit **isolé**.

Lexique (graphes orientés)

- Si (u,v) est un arc d'un graphe orienté $G = (S;A)$, on dit que (u,v) **part** du sommet u et **arrive** au sommet v .
- Dans un graphe orienté,
 - **le degré sortant** d'un sommet est le nombre d'arcs qui en partent,
 - **le degré entrant** est le nombre d'arcs qui y arrivent
 - **le degré** est la somme du degré entrant et du degré sortant.

Les chemins et les chaînes

- Dans un *graphe orienté* $G = (S;A)$, un **chemin de longueur k** d'un sommet u à un sommet v est une *séquence* $(u_0; u_1; \dots; u_k)$ de sommets telle que
 - $u = u_0$,
 - $v = u_k$
 - $(u_{i-1}, u_i) \in A$ pour tout i dans $\{1; \dots; k\}$.
- Un chemin est **élémentaire** si ces sommets sont tous distincts
- Pour un graphe non orienté, on parle de **chaîne** à la place de chemins

Sous chemins

- Un sous-chemin p' d'un chemin $p = (u_0; u_1; \dots; u_k)$ est une sous-séquence contiguë de ses sommets:
 - il existe i et j , avec $0 \leq i \leq j \leq k$, tels que
 - $p_0 = (u_i; u_{i+1}; \dots; u_j)$.
- Pour les graphes non orientés, on parle de la notion de sous chaîne.

Les circuits

- Dans un graphe $G=(S;A)$, un chemin $(u_0;u_1;...;u_k)$ forme **un circuit** si $u_0 = u_k$ et si le chemin contient au moins un arc.
- **Un circuit est élémentaire** si les sommets $u_1, ..., u_k$ sont distincts.
- Une boucle est un circuit de longueur 1
- Un graphe sans cycle est dit **acyclique**.

Connexité

- Un graphe non orienté est **connexe** si chaque paire de sommets est reliée par une chaîne.
- **Les composantes connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « est accessible à partir de »
- Un graphe orienté est **fortement connexe** si chaque sommet est accessible à partir de n'importe quel autre.
- **Les composantes fortement connexes** d'un graphe sont les classes d'équivalence de sommets induites par la relation « sont accessibles l'un à partir de l'autre ».



Arbres

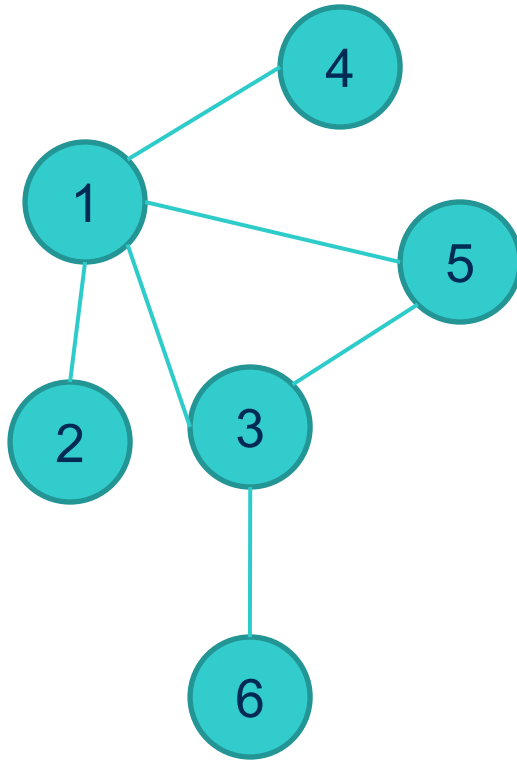
Définitions

- un graphe non orienté connexe acyclique est **un arbre**
- Un graphe non orienté acyclique est **une forêt**
- Les composantes connexes d'un graphe non orienté acyclique constituent les arbres de la forêt

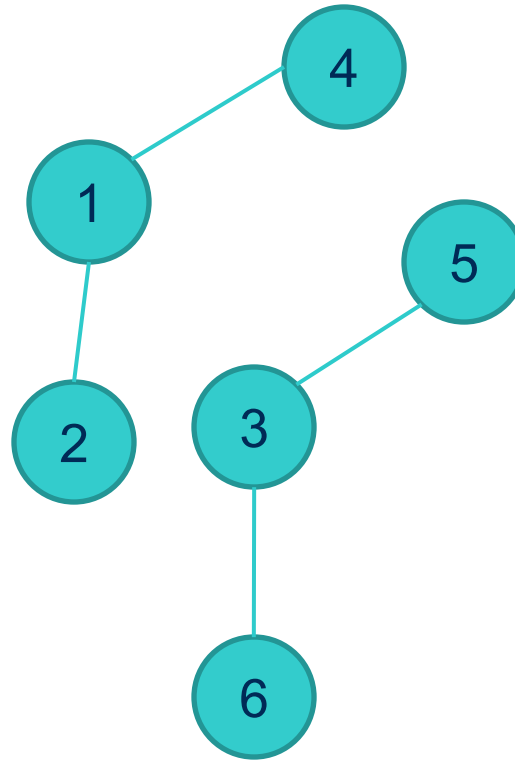
Définitions équivalentes

- Soit $G = (S;A)$ un graphe non orienté. Les affirmations suivantes sont équivalentes.
 - G est un arbre.
 - Deux sommets quelconques de G sont reliés par un unique chemin élémentaire.
 - G est connexe, mais si une arête quelconque est ôtée de A , le graphe résultant n'est plus connexe.
 - G est connexe et $|A| = |S|-1$.
 - G est acyclique et $|A| = |S|-1$.
 - G est acyclique, mais si une arête quelconque est ajoutée à A , le graphe résultant contient un cycle.

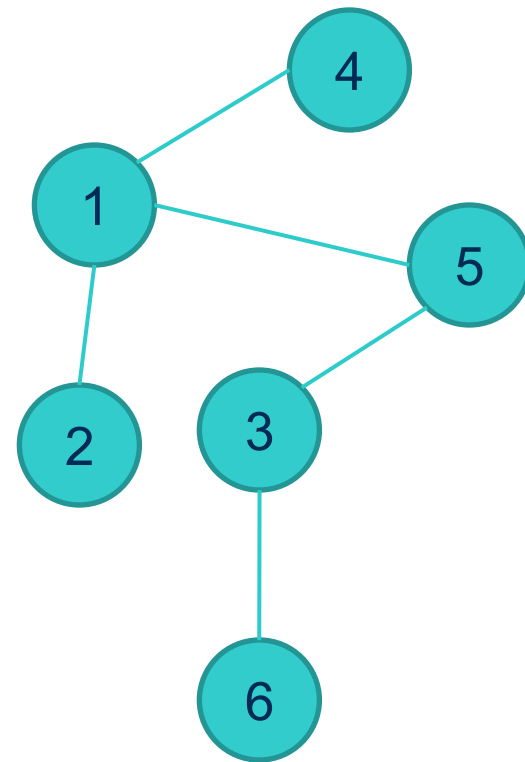
Exemples



Graphe contenant
un cycle



Forêt



Arbre

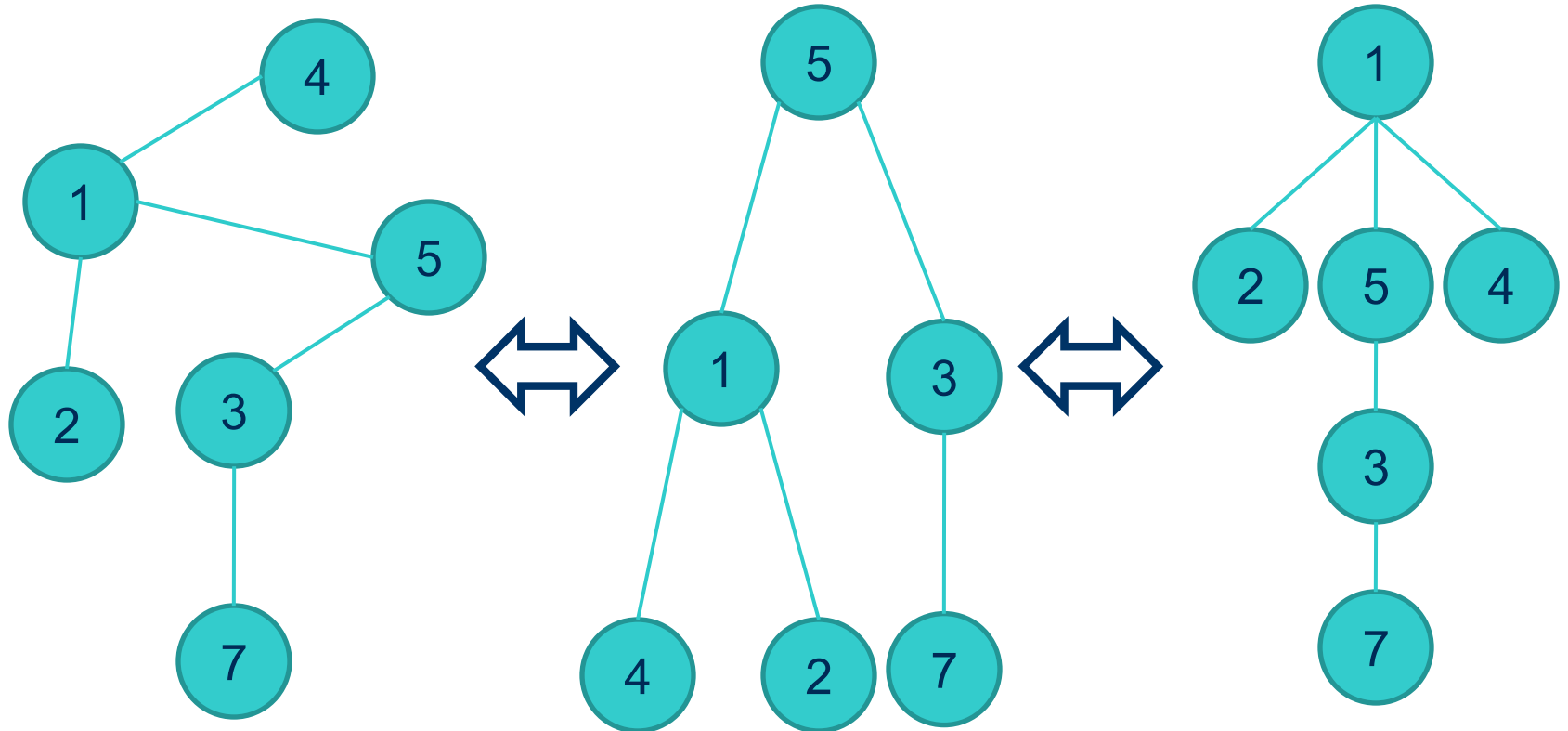
Arbre enraciné

- **Un arbre enraciné** est un arbre dans lequel l'un des sommets se distingue des autres.
- On appelle ce sommet **la racine**.
- Ce sommet particulier impose en réalité un sens de parcours de l'arbre et l'arbre se retrouve orienté par l'utilisation qui en est faite

Types de nœuds

- Soit x un nœud d'un arbre T de racine r . Un nœud quelconque y sur l'unique chemin allant de r à x est appelé **ancêtre** de x .
- Si T contient l'arête $\{y, x\}$ alors y est **le père** de x , et x est **le fils** de y .
- La racine est le seul nœud qui ne possède pas de père
- Un nœud sans fils est **un nœud externe** ou **une feuille**
- Un nœud qui n'est pas une feuille est **un nœud interne**
- Le **sous-arbre de racine x** est l'arbre composé des descendants de x , enraciné en x .

Exemples: deux arbres avec deux racines différentes



Graphe G

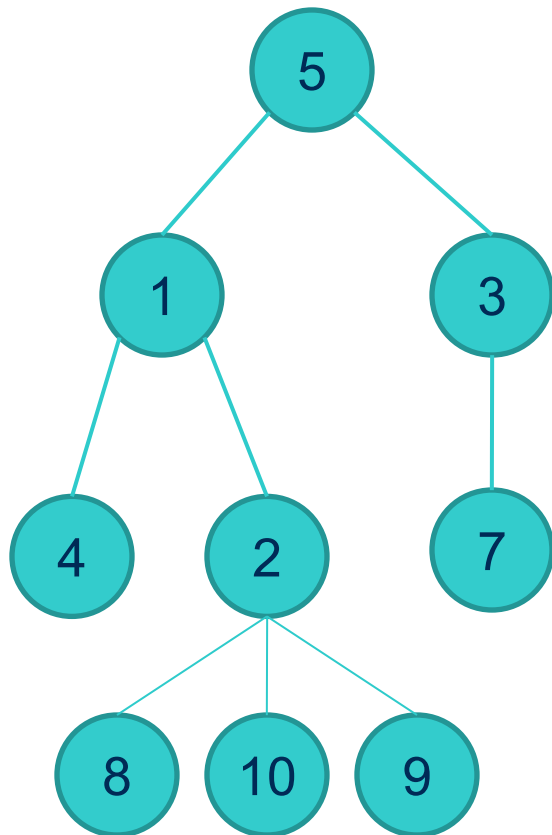
Arbre avec G enraciné
sur le nœud 5

Arbre avec G enraciné
sur le nœud 1

Degré et profondeur

- Le nombre de fils du nœud x est appelé le **degré de x**
- *Suivant qu'un arbre (enraciné) est vu comme un arbre ou un graphe, le degré de ses sommets n'a pas la même valeur*
- La longueur du chemin entre la racine r et le nœud x est la **profondeur de x**
- La plus grande profondeur que puisse avoir un nœud quelconque de l'arbre T est la **hauteur** de T
- Un **arbre ordonné** est un arbre enraciné dans lequel les fils de chaque nœud sont ordonnés entre eux.

Exemples profondeur et ordre

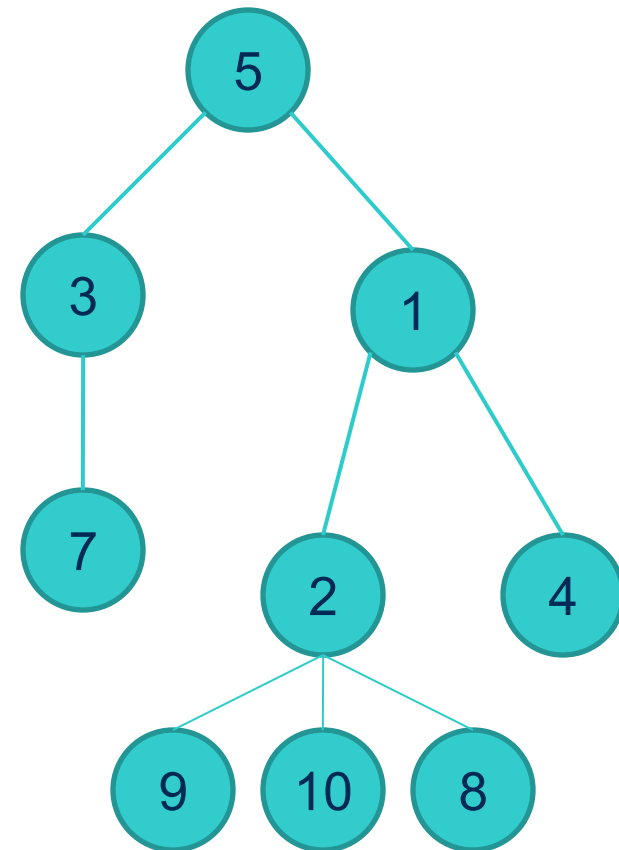


Profondeur 0

Profondeur 1

Profondeur 2

Profondeur 3



Arbres binaires

- Un **arbre binaire** est un arbre ordonné dont chaque nœud serait de degré au plus deux
- Un arbre binaire T est une structure définie récursivement sur un ensemble fini de nœuds et qui :
 - ne contient aucun nœud, ou
 - est formé de trois ensembles disjoints de nœuds:
 - une racine
 - un arbre binaire appelé son sous-arbre gauche
 - un arbre binaire appelé son sous-arbre droit
- Dans un arbre binaire, si un nœud n'a qu'un seul fils, la position de ce fils (fils gauche ou fils droit) est importante

Arbres complets

- Dans un **arbre binaire complet** chaque nœud est soit une feuille, soit de degré deux, aucun nœud n'est donc de degré un.
- Un **arbre k-aire** est une généralisation de la notion d'arbre binaire où chaque nœud est de degré au plus k et non plus simplement de degré au plus 2
- Un **arbre k-aire complet** est un arbre où chaque nœud est soit de degré k ou soit une feuille

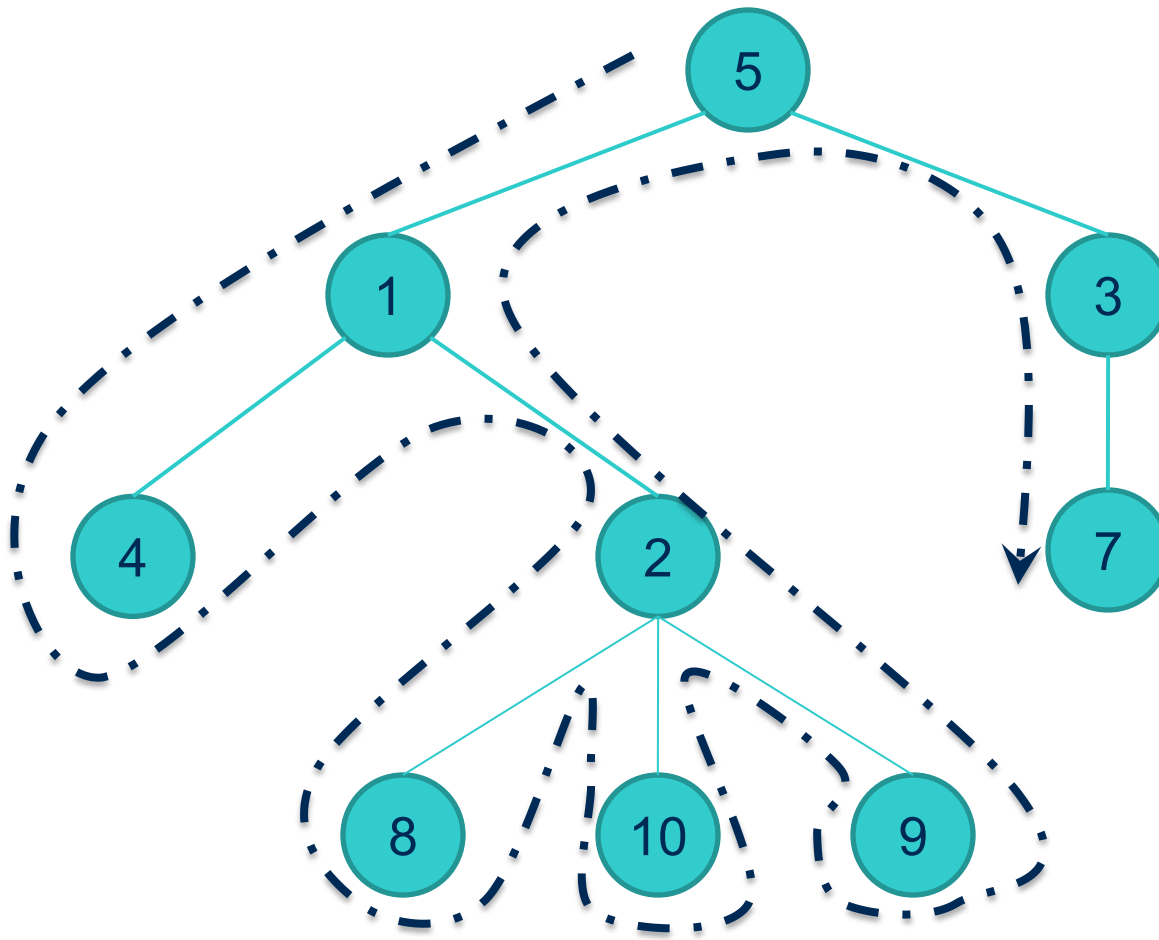


Parcours des arbres ordonnés

Parcours en profondeur

- Dans un *parcours en profondeur d'abord*:
 - on descend le plus profondément possible dans l'arbre
 - une fois qu'une feuille a été atteinte, on remonte pour explorer les autres branches en commençant par la branche la plus basse parmi celles non encore parcourues
 - les fils d'un nœud sont bien évidemment parcourus suivant l'ordre sur l'arbre

Sens du parcours



Algorithme de parcours en profondeur d'abord

- AfficherParcoursProfondeur(A)

Affiche racine(A)

si A n'est pas réduit à une feuille

alors

pour tous les fils u de $\text{racine}(A)$ (dans l'ordre)

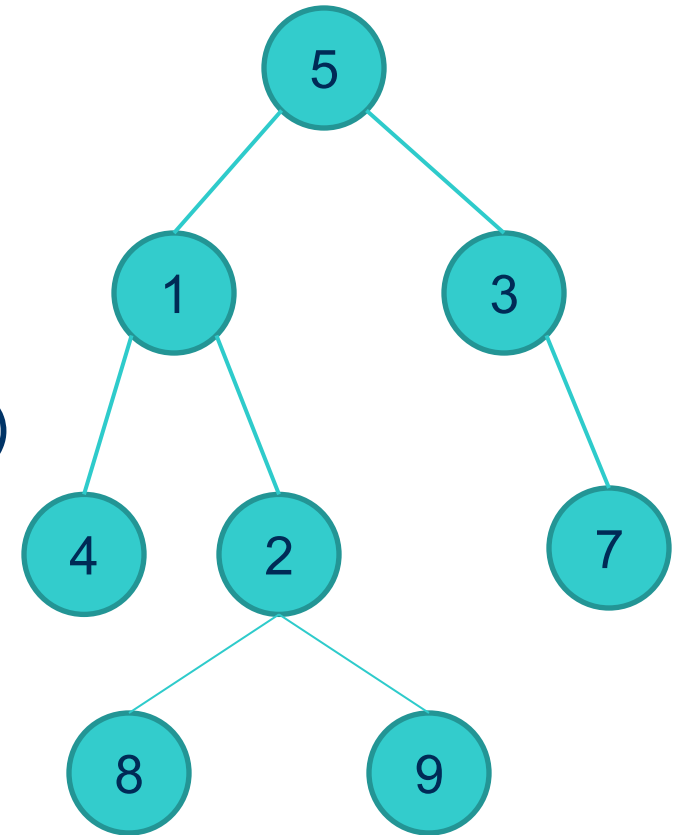
ParcoursProfondeur(u)

FinPour

Finsi

Exemples d'utilisation de la recherche en profondeur

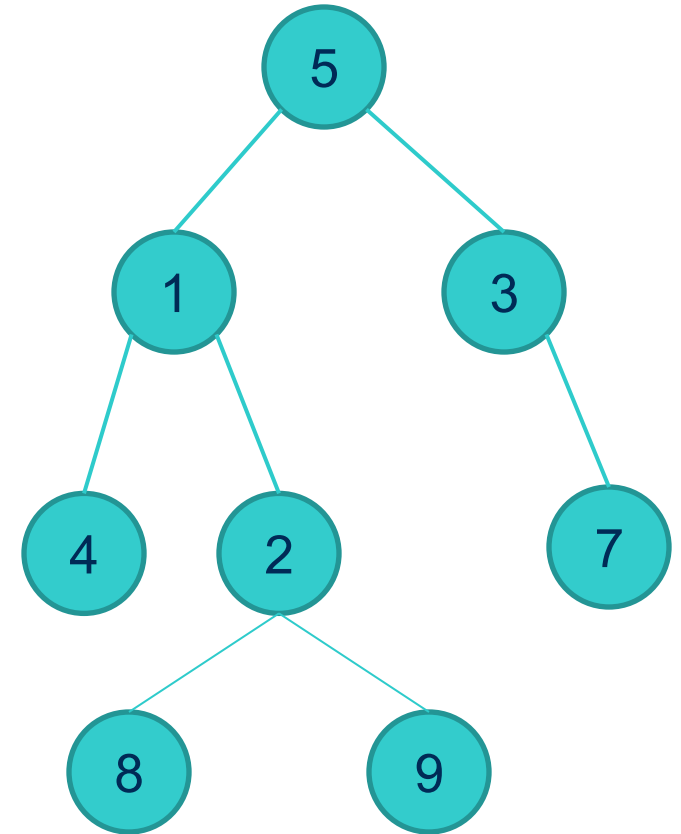
- Algo1: cas arbres binaires
 - PRÉFIXE(A)
si $A \neq \text{NIL}$ alors
affiche racine(A)
PRÉFIXE(FILS-GAUCHE(A))
PRÉFIXE(FILS-DROIT(A))



Résultat: 5;1;4;2;8;9;3;7

Exemples d'utilisation de la recherche en profondeur

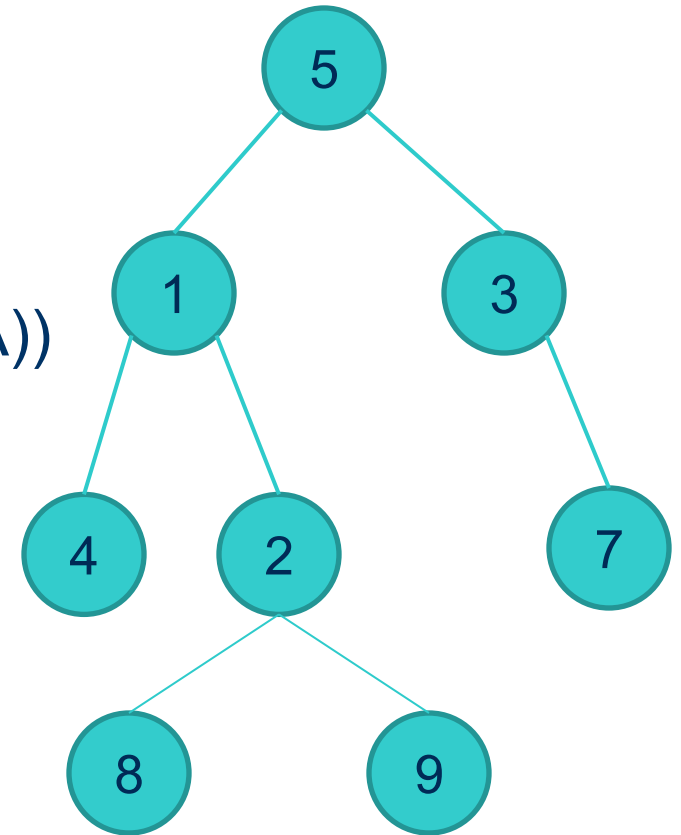
- Algo2: cas arbres binaires
 - INFIXE(A)
si $A \neq \text{NIL}$ faire
 INFIXE(FILS-GAUCHE(A))
 affiche racine(A)
 INFIXE(FILS-DROIT(A))



Résultat: 4;1;8;2;9;5;3;7

Exemples d'utilisation de la recherche en profondeur

- Algo3: cas arbres binaires
 - POSTFIXE(A)
si $A \neq \text{NIL}$ faire
 POSTFIXE(FILS-GAUCHE(A))
 POSTFIXE(FILS-DROIT(A))
 affiche racine(A)



Résultat: 4;8;9;2;1;7;3;5

Parcours en largeur

- Dans un *parcours en largeur d'abord*:
 - On commence par la racine
 - On parcourt tous les nœuds à la profondeur i dans l'ordre
 - On passe ensuite au nœuds de profondeur $i+1$
 - Jusqu'à atteindre la profondeur de l'arbre

Algorithme de parcours

- Pour établir l'algorithme de parcours en largeur d'abord, il faut sauvegarder les branches non encore visitées
- La structure la plus appropriée pour cela est la file
- ParcoursLargeur(A)

$F \leftarrow \{\text{racine}(A)\}$

tant que $F \neq \{\}$ faire

$u \leftarrow \text{SUPPRESSION}(F)$

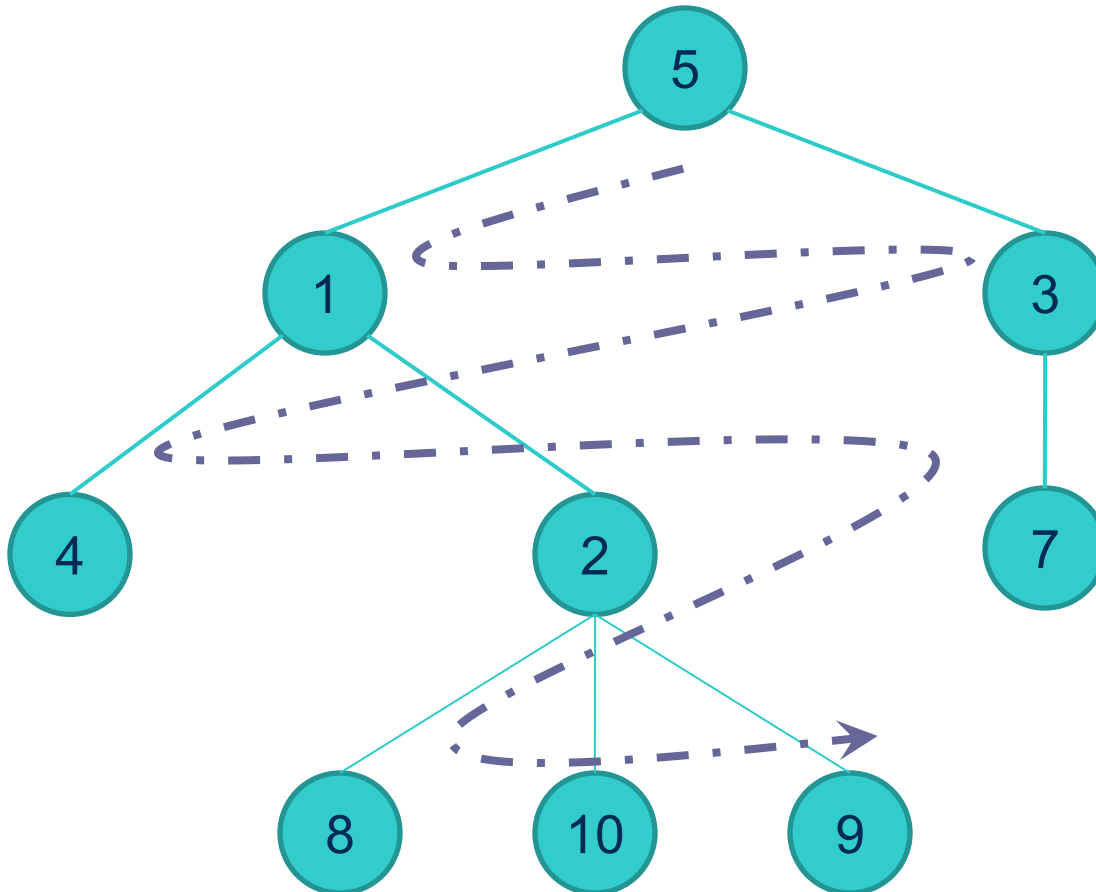
 pour tous les fils v de u faire dans l'ordre

$\text{INSERTION}(F, v)$

 FinPour

FinTantQue

Sens du parcours



Parcours des graphes

Technique de coloriage

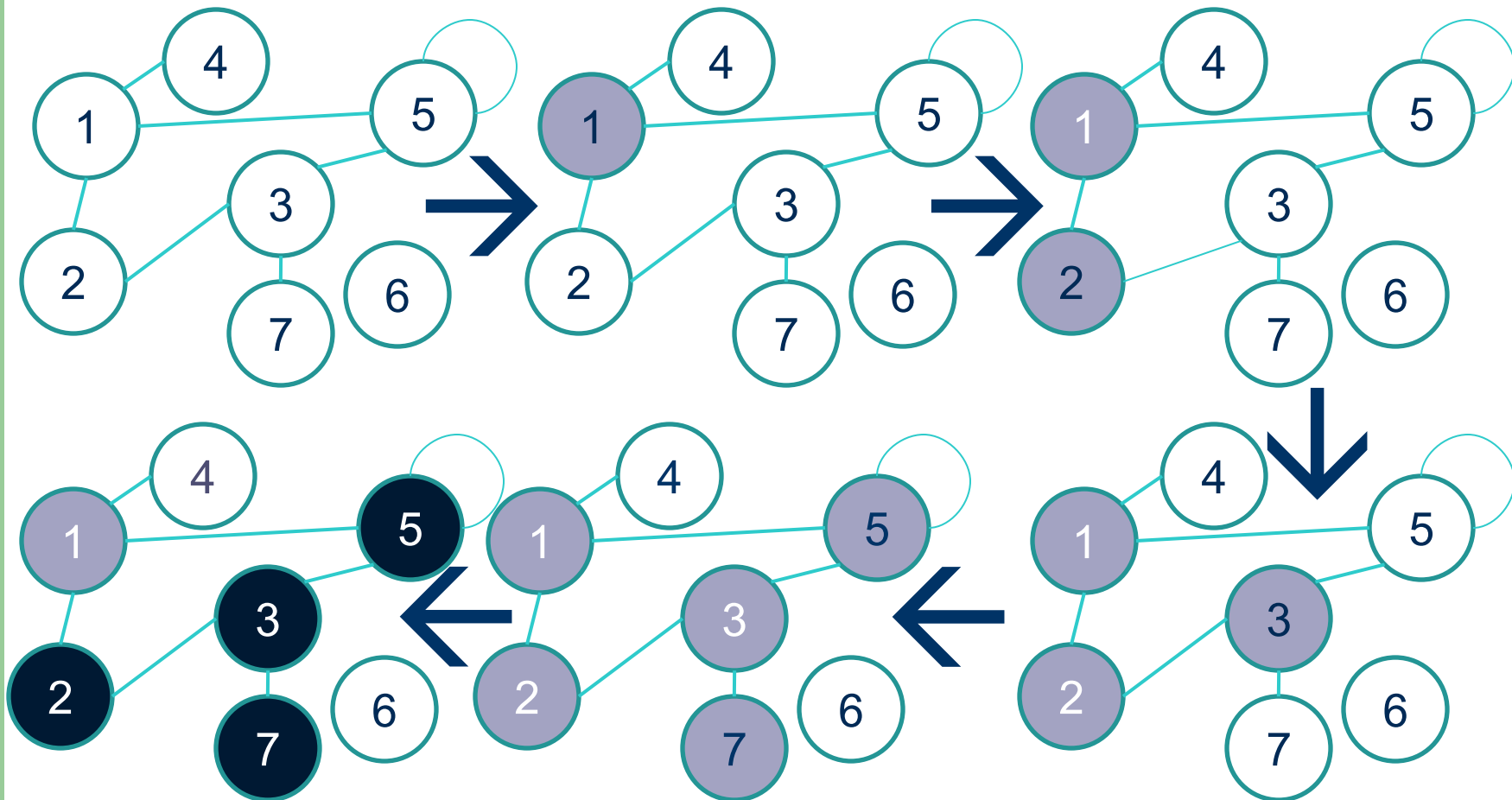
- Le parcours des graphes est plus complexe que le parcours des arbres
 - Les cycles!
- Objectif: éviter de boucler indéfiniment à l'intérieur d'un cycle
- Technique classique utilisée: colorier les nœuds
 - Initialement, les nœuds sont coloriés en blancs
 - Une fois visité, un nœud est colorié en gris
 - Une fois tous les fils d'un nœud sont visités, il est colorié en noir

Parcours en profondeur

```
ParcoursProfondeur(G)
  pour chaque sommet u de G faire
    couleur[u] ← BLANC
  FinPour
  pour chaque sommet u de G faire
    si couleur[u] = BLANC
      alors
        VISITER-Profondeur(G, u)
      FinSi
  FinPour
```

```
VISITER-Profondeur(G, s)
  couleur[s] ← GRIS
  Afficher(s)
  pour chaque voisin v de s faire
    si couleur[v] = BLANC
      alors
        VISITER-Profondeur(G, v)
      FinSi
  FinPour
  couleur[s] ← NOIR
```

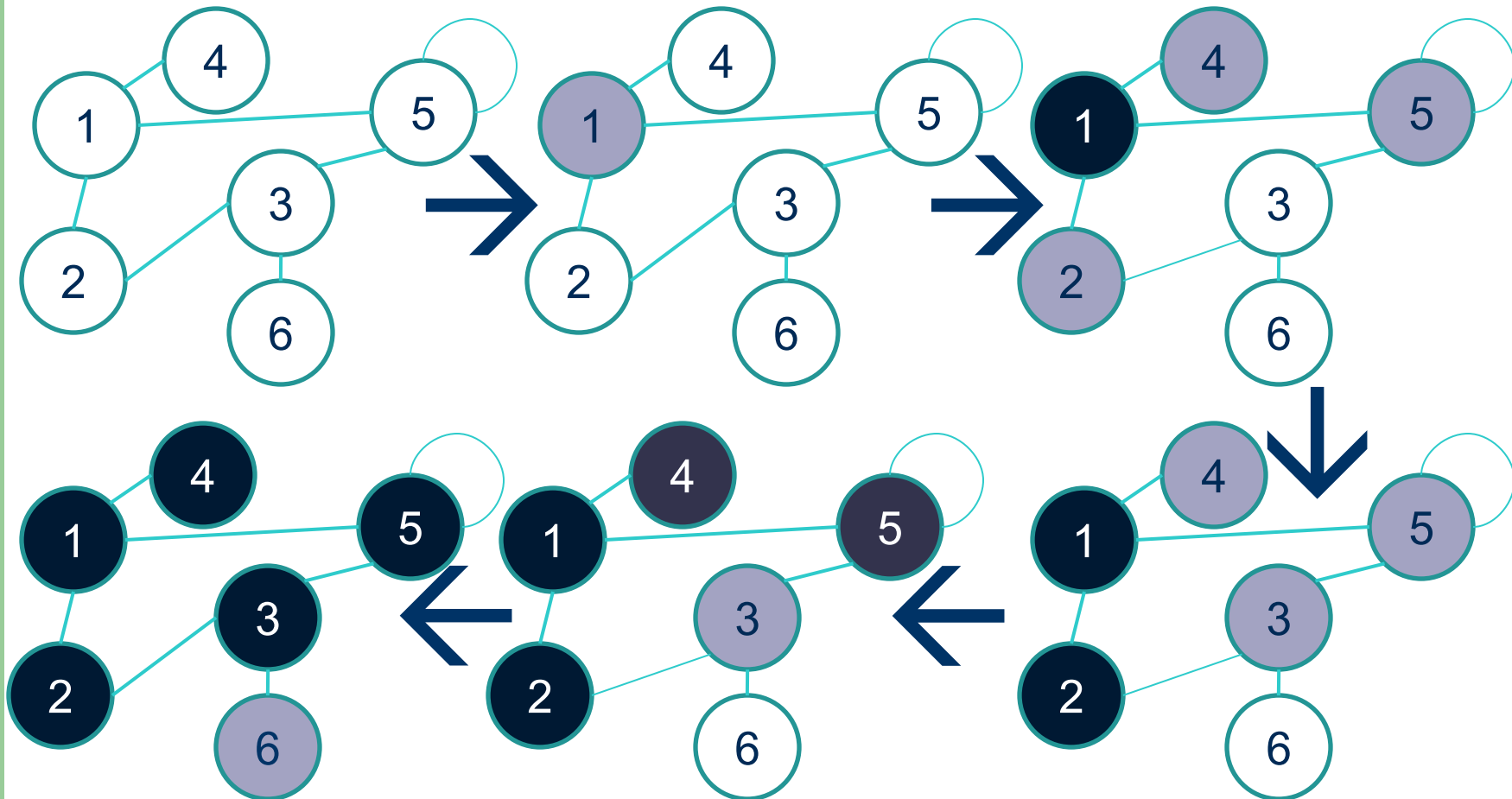
Exemple: En profondeur



Parcours en largeur

- `ParcoursLargeur(G, s)`
 `couleur[s] ← GRIS`
 Pour chaque sommet `u` de `G` avec `u <> s` faire `couleur[u] ← BLANC`
 FinPour
 `F ← {s}`
 tant que `F <> { }` faire
 `u ← SUPPRESSION(F)`
 pour chaque voisin `v` de `u` faire
 si `couleur[v] = BLANC`
 alors `couleur[v] ← GRIS`
 `INSERTION(F;v)` , FinSi
 FinPour
 `couleur[u] ← NOIR`
 FinTantQue

Exemple: En largeur

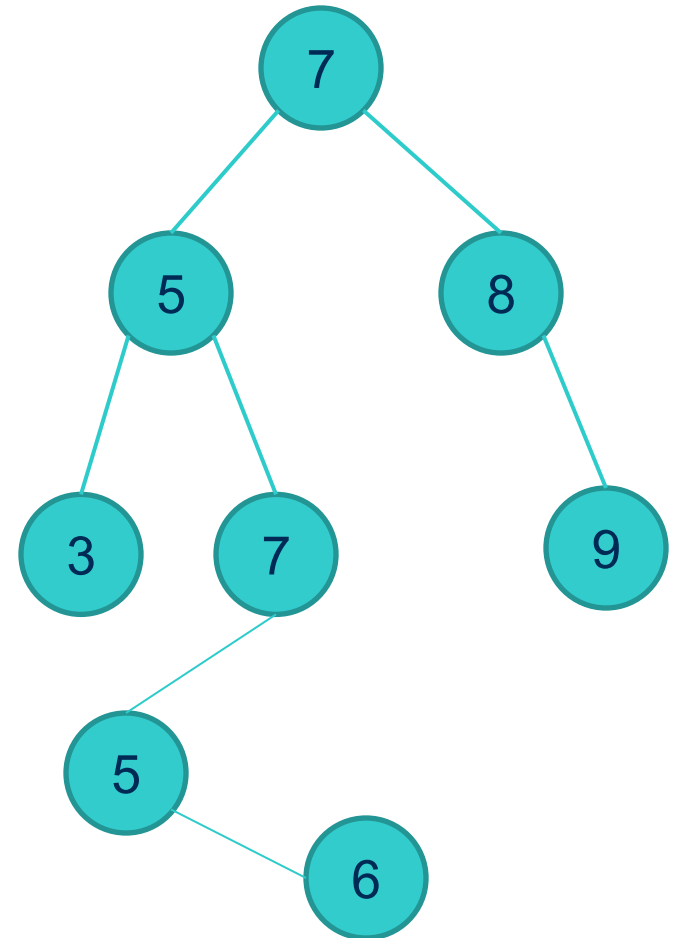
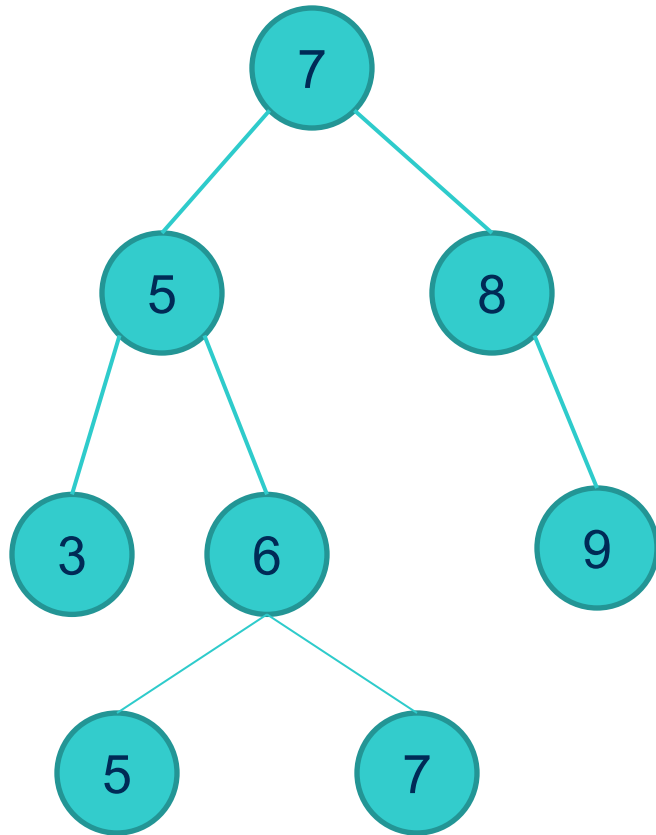


Arbres de recherche

Arbre binaire de recherche

- *Un arbre binaire de recherche est un arbre binaire vérifiant:*
 - *soient x et y deux nœuds de l'arbre,*
 - *si y est un nœud du sous-arbre gauche de x , alors $\text{clef}(y) \leq \text{clef}(x)$,*
 - *si y est un nœud du sous-arbre droit de x , alors $\text{clef}(y) \geq \text{clef}(x)$.*

Exemple: deux arbres binaires de recherche



Algorithmes de recherche



Recherche d'élément

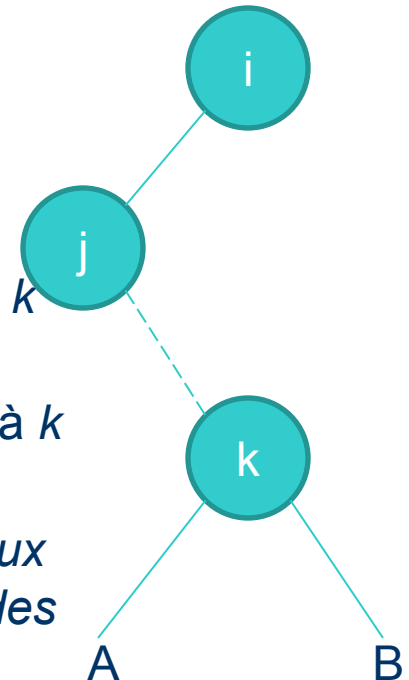
- ARBRE-RECHERCHER(x , k)
 - si $x = \text{NIL}$ ou $k = \text{clé}(x)$ alors
 - renvoyer x
 - FinSi
 - si $k < \text{clé}(x)$ alors
 - renvoyer ARBRE-RECHERCHER($\text{gauche}(x)$, k)
 - sinon
 - renvoyer ARBRE-RECHERCHER($\text{droit}(x)$, k)
 - FinSi

Recherche de minimum

- ARBRE-MINIMUM(x)
tant que gauche(x) \neq NIL faire
 $x \leftarrow$ gauche(x)
FinTantQue
renvoyer x
- ARBRE-MAXIMUM(x)
tant que droit(x) \neq NIL faire
 $x \leftarrow$ droit(x)
FinTantQue
renvoyer x

Recherche du successeur

- A sous arbre gauche / B sous arbre droit
- j, le plus proche ancêtre qui est fils gauche (k si lui-même fils gauche) / i, le père de j
- Propriétés:
 - Le sous-arbre A ne contient que des clés inférieures à k (*ne peuvent être successeurs*)
 - Le sous-arbre B ne contient que des clés supérieures à k *et peut contenir le successeur de k s'il n'est pas vide.*
 - Tous les ancêtres de k jusqu'à j sont inférieurs ou égaux à k *et leurs sous-arbres gauches ne contiennent que des valeurs inférieures à k.*
 - La valeur de i est supérieure à toutes celles contenues dans son sous-arbre gauche (de racine j) et donc à k et à celles de B. Toutes les valeurs de son sous-arbre droit sont supérieures à i.



Conclusion et algorithme

- Conclusion:
 - si B est non vide, son minimum est le successeur de k ,
 - *sinon le successeur de k est le premier ancêtre de k dont le fils gauche est aussi ancêtre de k .*
- **ARBRE-SUCCESSEUR(k)**
 - si droit(k) \neq NIL alors renvoyer ARBRE-MINIMUM(droit(k))*
 - FinSi*
 - $j \leftarrow \text{père}(k)$*
 - tant que $j \neq$ NIL et $k = \text{droit}(j)$ faire*
 - $k \leftarrow j$*
 - $j \leftarrow \text{père}(j)$*
 - FinTantQue*
 - renvoyer j*

Recherche du prédécesseur

- Faisons le ensemble!



FIN

